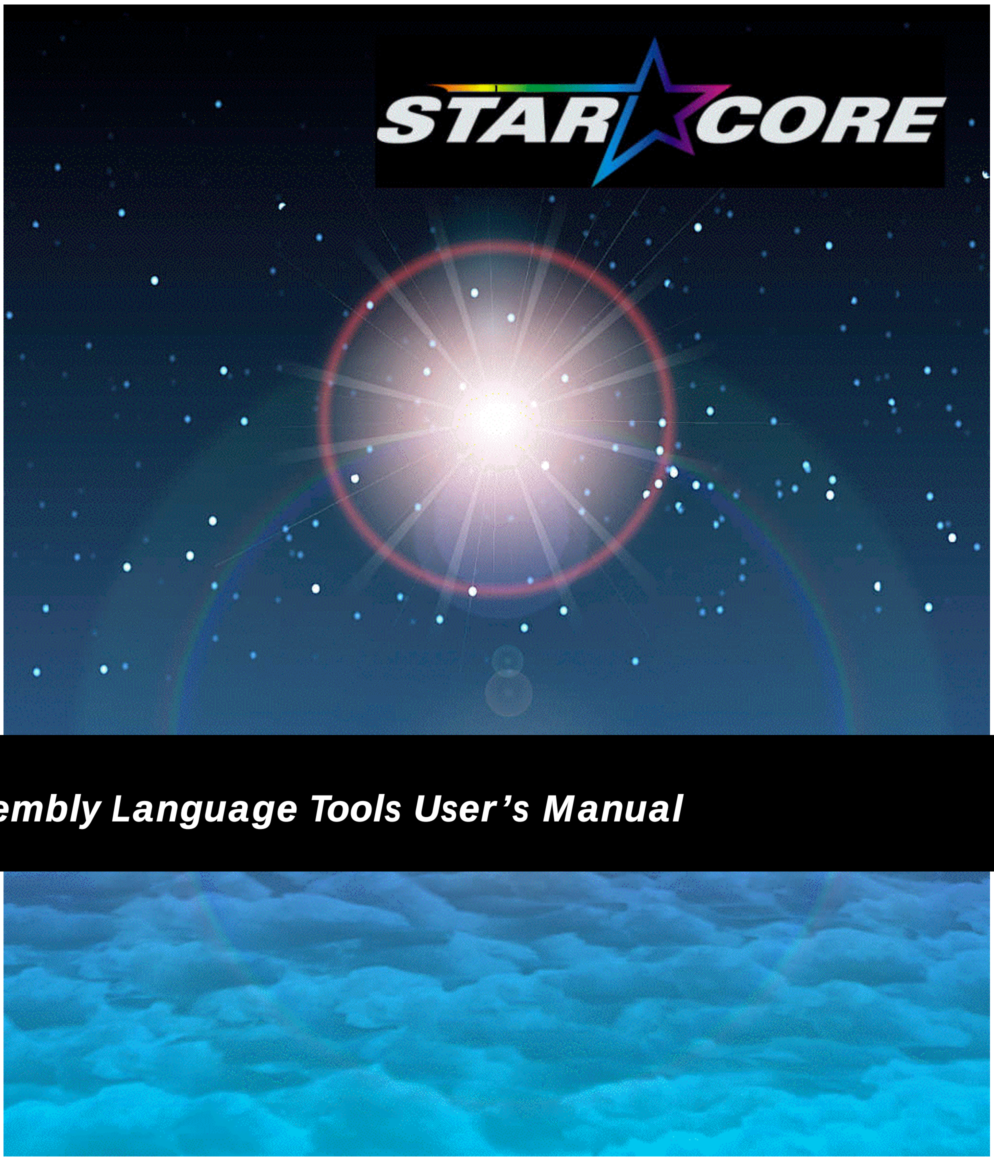
MNSC100ALT/D
Rev. 1.5, 8/2000



SC100 Assembly Language Tools User's Manual

microelectronics group

Lucent Technologies
Bell Labs Innovations

DigitalDNA™
from Motorola

MNSC100ALT/D
Rev. 1.5, 8/2000

SC100 Assembly Language Tools

User's Manual

STAR CORE
BRIGHTER DSP TECHNOLOGY!

Digital DNA™ from Motorola

microelectronics group
Lucent Technologies
Bell Labs Innovations

Table of Contents

List of Tables

List of Figures

List of Examples

About This Book

This manual describes how to use the SC100 assembly language tools, which consist of an assembler, linker, archiver, and various utilities.

Audience

This manual is intended for systems software developers, applications programmers, and microprocessor designers.

Organization

This manual is organized as follows:

- Chapter 1, "Introduction," provides an overview of the SC100 assembly language tools.
- Chapter 2, "SC100 Assembler," describes the SC100 Assembler and how to invoke it. This chapter also discusses various components of assembler source statements and the source listing file.
- Chapter 3, "Expressions," describes the expressions accepted as operands in assembler instructions or directives. This chapter also describes the built-in functions supported by the assembler and provides examples of their use.
- Chapter 4, "Software Project Management," discusses the assembler directives designed to assist in the development of large software projects.
- Chapter 5, "Assembler Directives and Significant Characters," describes the directives which direct the behavior of the assembler, and the special characters that are significant to it. Each description includes an example.
- Chapter 6, "Macro Operations and Conditional Assembly," discusses assembler macros and conditional assembly.
- Chapter 7, "SC100 Linker," describes the SC100 Linker and how to invoke it.
- Chapter 8, "SC100 Utilities," describes the SC100 utilities and how to invoke them.
- Appendix A, "ASCII Character Codes,“ lists ASCII character codes, and their decimal and hexadecimal equivalents.
- Appendix B, "Assembler Messages,“ describes the assembler error and warning messages.

References

- *SC100 Application Binary Interface Reference Manual* (MNSC100ABI/D)
- *SC100 C Compiler User's Manual* (MNSC100CC/D)
- *SC140 DSP Core Reference Manual* (MNSC140CORE/D)
- *SC100 Simulator Reference Manual* (MNSC100SIM/D)

Conventions

The following notational conventions are used in this document:

- Courier monospaced type indicates commands, command parameters, and code examples:
  - — Bold type indicates the elements of command lines that must be entered exactly as shown.
  - — Italic type indicates replaceable command parameters that the user must provide.
- Command syntax elements used in this document include the following:

{ }	Braces contain a list of choices, from which the user must choose one. Each choice is separated by a vertical bar. For example, `{R|L}` indicates that either R or L must be selected.
[ ]	Square brackets contain one or more optional items. If more than one optional item is shown, the required element separators are indicated. For example, the syntactical element [*`number`*,] requires the comma to be specified if `number` is selected.
...	An ellipsis indicates that an argument can be repeated several times in a command line.

- All assembler command-line options, mnemonics, and directives are shown in upper case to highlight them. However, the assembler recognizes both upper and lower case for these elements.

Chapter 1
Introduction

From the outset, the StarCore Technology Center has focused on ensuring a wide selection of best-in-class development tools for StarCore-based SoC products. The result is an unusually high level of support for a new architecture that includes multiple compilers, development environments, and real-time operating system software products.

Specifically, StarCore is developing baseline tools such as an assembler, linker, C compiler, and simulator. These common SC100 baseline tools will be featured in visually integrated development environments (IDEs) that will be provided by Lucent Technologies and Motorola in support of their respective SC100 chip products. The IDEs will include real-time source-level debugging and profiling tools.

This manual describes the SC100 assembly language tools, which include an assembler, a linker, and several utilities.

For information on the C compiler and simulator, refer to the *SC100 C Compiler User's Manual* and *SC100 Simulator Reference Manual*, respectively.

1.1 Assembler

The SC100 Assembler converts handwritten or compiler-generated SC100 assembly code into ELF object files. Key features of the assembler include the following:

- Expression evaluation using numeric and string constants, operators, and built-in functions
- Modular programming using sections
- Macros that allow variable arguments
- Conditional assembly

A complete description of the assembler is provided in Chapters 2 through 6 of this manual.

1.2 Linker

The SC100 Linker combines multiple relocatable object files and archive files, relocates their data, adjusts the symbol references, and generates an executable object file. The linker allows programmers to break up a large program into more manageable modules that can be assembled separately. This is useful when troubleshooting an application. Only the module with a problem needs to be edited and reassembled. The updated object module can then be relinked with the other object modules to produce a new executable object file.

For a complete description of the linker and its use, refer to Chapter 7, "SC100 Linker."

1.3 Utilities

Included with the StarCore software development tools are several utilities that process or interpret ELF object files. These utilities are listed in Table 1-1, and are described in Chapter 8, "SC100 Utilities."

Table 1-1. SC100 Utilities

Utility	Name	Function
`arsc100`	Archiver	Groups separate object files together into a single file for linking or archival storage
`sc100-dis`	Disassembler	Disassembles executable instructions contained in object files
`sc100-elfdump`	ELF File Dump	Outputs a formatted display of the contents of object files
`sc100-nm`	Name	Lists the symbolic information for object files
`sc100-size`	Size	Lists the size of sections and segments contained in object files

1.4 Software Development Flow

Figure 1-1 illustrates the software development flow, and shows the inputs and outputs of each stage.

C source files are submitted to the compiler. By default, the compiler shell automatically advances the source files through the compilation, assembly, and linking stages and produces an executable program. Compiler options allow the programmer to select which development tools and processing stages are invoked, and to define any processing options or settings that are needed.

Hand-coded assembly language files are submitted to the assembler, which transforms the assembly code into ELF object code. The object code and any object modules are then submitted to the linker, which combines them into a single, executable program.

Finally, the executable program may be loaded into the simulator where it can be executed and evaluated.

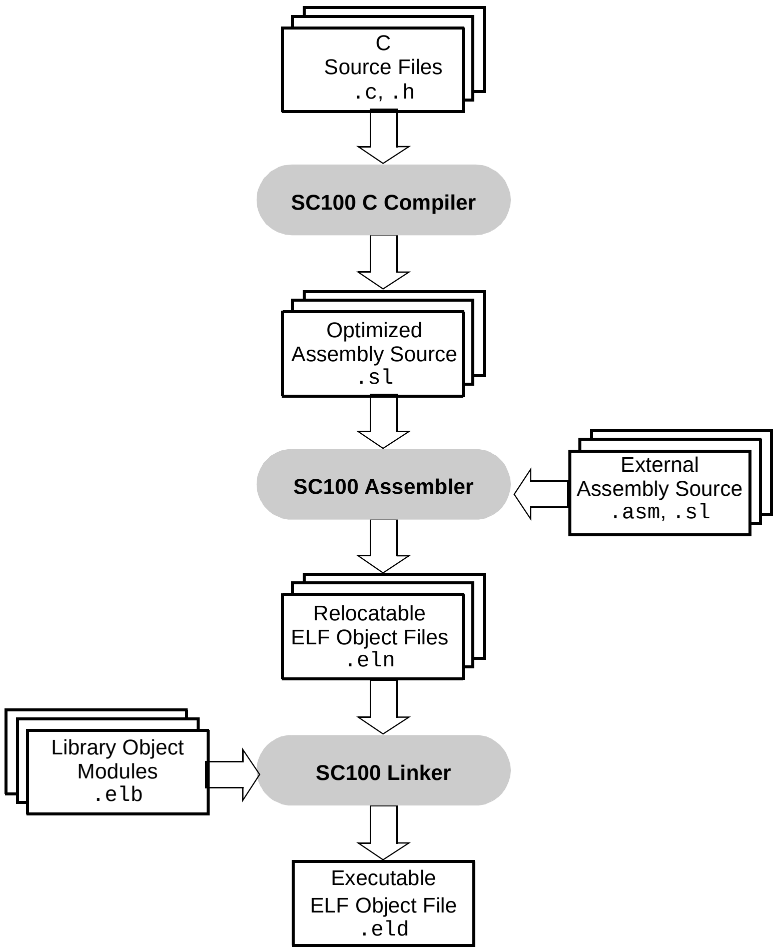


Figure 1-1. StarCore Development Tools

Chapter 2 SC100 Assembler

The SC100 Assembler translates hand-written or compiler-generated SC140 assembly language programs into machine language, using the executable and linking format (ELF) for object files. The assembler supports the following features:

- Expression evaluation using numeric and string constants, operators, and built-in functions
- Modular programming using sections
- Macros that allow variable arguments
- Conditional assembly

2.1 Invoking the Assembler

The command line to invoke the assembler is as follows:

`asmsc100` `[`*`option ...`*`]` *`file ...`*

where:

`option`	One or more of the options listed in Table 2-1 on page 2-2. Assembler options are not case sensitive.
`file`	One or more assembly source files. Multiple file names must be separated by spaces. The files are processed in the order listed, and are assembled into a single object file. If no extension is supplied with the file name, the assembler first attempts to open the file using the name as entered. If that is not successful, the assembler appends `.asm` to the name, and attempts to open the file again.

If invoked with no options, the assembler processes the source file, and outputs a source listing to the standard output. An object file is not generated by default; one must be specifically requested using the `-b` option. See Section 2.2.3, “Generating an Object File,” for details.

Example 2-1 shows a basic command that invokes the assembler, using all default settings except that of generating an object file. This command assembles the source file `corr.asm`, outputs a source listing to the standard output, and generates a relocatable object file named `corr.eln`.

Example 2-1. Basic Command to Invoke Assembler

```
asmsc100 -b corr.asm
```

2.2 Assembler Command-Line Options

Table 2-1 summarizes the options available with the assembler. These options are discussed in further detail in the sections that follow.

Table 2-1. SC100 Assembler Options

Option	Description	Section	Page
-a	Directs the assembler to operate in absolute mode instead of the default relative mode. This option generates an executable object file when used with the `-b` option.	2.2.3	2-4
-b[*objfile*]	Generates an object file and assigns the specified name to it. No space is allowed between `-b` and the file name.	2.2.3	2-4
-d *symbol string*	Defines substitution strings to be used on all source lines. This option is equivalent to the DEFINE directive.	2.2.8	2-6
-ea *errfil*	Appends the standard error output stream to the specified file. A space is required between `-ea` and the file name.	2.2.6	2-5
-ew *errfil*	Writes the standard error output stream to the specified file, overwriting the file if it already exists. A space is required between `-ew` and the file name.	2.2.6	2-5
-f*argfil*	Reads options and file names from the specified file, and appends them to the command line.	2.2.2	2-3
-g	Adds debug information to the object file. This option is valid only when used with the `-b` option.	2.2.4	2-4
-i*pathname*	Adds the specified directory to the standard search paths. The `-i` option may be repeated as many times as desired. The directories are searched in the order that they are listed on the command line.	2.2.7	2-6
-l[*lstfil*]	Redirects the source listing to the specified file. No space is allowed between `-l` and the file name.	2.2.5	2-5
-m*directory*	Specifies the directory that contains macro definitions. The `-m` option may be repeated as many times as desired. The directories are searched in the order that they are listed on the command line. This option is equivalent to the MACLIB directive.	2.2.7	2-6
-o*opt*[,*opt*...]	Designates one or more assembler options. Multiple options must be separated by commas with no intervening spaces. Valid `opt` arguments are any of the OPT options listed in Table 5-3 on page 5-57. This option is equivalent to the OPT directive.	2.2.9	2-7
-q	Quiet mode. Suppresses the assembler banner.	2.2.6	2-5
-s {**all**|*tag*[,*tag*...]}	Enables restriction error checking, which is off by default. A space is required between `-s` and the first argument. Multiple arguments must be separated by commas with no intervening spaces. Valid `tag` arguments are listed in Table 2-2 on page 2-8.	2.2.11	2-7
-v	Verbose mode. Reports assembly progress.	2.2.6	2-5
-z	Strips symbol information from the executable object file. This option is valid only when used with the `-a` and `-b` options.	2.2.4	2-4

2.2.1 Using an Environment Variable

Command-line options that are used regularly may be assigned to the environment variable DSPASMOPT. The assembler adds the text associated with this variable to the existing command line prior to processing any options.

To define DSPASMOPT, add the command line below that is appropriate for your operating system to the `env.*` environment file located in the tools installation directory. Re-execute the `env.*` file when finished.

Operating System	Command line	Environment File

UNIX:
Bourne shell (sh, ksh, bash)
	`DSPASMOPT="-`*`option`*` ..."`
	`export DSPASMOPT`
		`env.sh`

UNIX: C shell (csh, tcsh)	`setenv DSPASMOPT "-`*`option`*` ..."`	`env.csh`
Windows	`set DSPASMOPT=-`*`option`*` ...`	`env.bat`

where: *option* is any option listed in Table 2-1. Multiple options must be separated by a space, and each option must be preceded by a hyphen.

For example, assume DSPASMOPT is defined as follows in the `env.sh` file:

```
DSPASMOPT="-b -l"
export DSPASMOPT
```

Each time the assembler is invoked, it will add the `-b` and `-l` options to the command line. Therefore, the command: `asmsc100 corr.asm`
will be transformed by the assembler into: `asmsc100 -l -b corr.asm`

2.2.2 Reading Input from an Argument File

Command-line input from an argument file can be passed to the assembler using the `-f` option, as follows:

`-f`*`argfile`*	Instructs the assembler to read command-line input from the named argument file. The file name can include an optional pathname.

The argument file is a text file containing a list of options, arguments, file names, or the `-f` option itself. Within the argument file, each separate file or option (with or without an argument) must be separated by some form of white space, such as a blank, tab, or newline. Comments can be included in the file using a semicolon (;). All characters following the semicolon are ignored by the assembler.

The command in Example 2-2 invokes the assembler, reading arguments from the file `asmopts`.

Example 2-2. Specifying an Argument File

```
asmsc100 -fasmopts filter.asm
```

2.2.3 Generating an Object File

The assembler generates an object file only when the `-b` option is specified on the command line. The object file is relocatable, by default. If the `-a` option is specified on the command line with `-b`, the resulting object file is executable.

`-a`	Generates an executable object file when used with the `-b` option.
`-b [`*`file`*`]`	Generates an object file and assigns the specified name to it. A space is not allowed between `-b` and the file name. The file name may include an optional pathname. If a hyphen is used in place of a file name, the object file is sent to the standard output. Any file with the same name is overwritten. If no name is specified, the object file is given the same name as the first source file encountered on the command line, with an extension of `.eln` if the object file is relocatable, or `.eld` if it is executable.

The command shown in Example 2-3 assembles two assembly files, `main.asm` and `fft.asm`, into an executable object file named `filter.eld`.

Example 2-3. Generating an Executable Object File

```
asmsc100 -a -bfilter.eld main.asm fft.asm
```

2.2.4 Modifying the Contents of the Object File

The options in this section control whether certain types of information are included in the object file.

`-g`	Adds the following debugging sections to the object file: .debug_abbrev, .debug_aranges, .debug_info, and .debug_line. The `-g` option is valid only when used with the `-b` option.
`-z`	Strips symbol information from the executable object file. This option is valid only when used with both the `-a` and `-b` options.

The command in Example 2-4 strips symbol information from the executable object file, `filter.eld`.

Example 2-4. Stripping Symbol Information from the Object File

```
asmsc100 -a -bfilter.eld -z filter.asm
```

2.2.5 Redirecting the Source Listing

By default, the assembler sends a source listing to the standard output. The source listing can be saved to a file using the `-l` option.

`-l[`*`file`*`]`	Redirects the source listing to the specified file. The file name may include an optional pathname. A space is not allowed between `-l` and the file name. Any file with the same name is overwritten. If no name is specified, the listing is given the same name as the first source file encountered on the command line, and an extension of `.lst`.

The command shown in Example 2-5 assembles the files `filter.asm` and `gaus.asm` into a single, relocatable object file named `filter.eln`, and redirects the source listing to a file named `filter.lst`.

Example 2-5. Redirecting the Source Listing to a File

```
asmsc100 -b -lfilter.lst filter.asm gaus.asm
```

2.2.6 Controlling Output Messages

The options in this section redirect the standard error output stream to a file, and control the level of informational messages displayed by the assembler.

`-ea `*`file`*	Appends the standard error output stream, stderr, to the specified file. A space is required between `-ea` and the file name.
`-ew `*`file`*	Writes the standard error output stream, stderr, to the specified file, overwriting the file if it already exists. A space is required between `-ew` and the file name.
`-q`	Quiet mode. Suppresses the assembler banner.
`-v`	Verbose mode. Reports assembly progress (such as beginning of passes, and opening and closing of input files) to the standard error output stream. This is useful to insure that assembly is proceeding normally.

The command shown in Example 2-5 assembles the files `filter.asm` and `gaus.asm` into a relocatable object file named `filter.eln`, redirects the standard error output stream to the file `errors`, and redirects the source listing to a file named `filter.lst`.

Example 2-6. Redirecting the Source Listing to a File

```
asmsc100 -b -ew errors -lfilter.lst filter.asm gaus.asm
```

2.2.7 Adding Directories to the Search Path

The options in this section add directories to the standard search paths used by the assembler.

`-i`*`pathname`*	Adds the specified directory to the path used by the assembler to search for INCLUDE files.
`-m`*`pathname`*	Adds the specified directory to the path used by the assembler to search for macro definitions. This option is equivalent to the MACLIB directive.

Both the `-i` option and the `-m` option may be repeated as many times as desired. The directories will be searched in the order specified on the command line.

Example 2-7. Adding Directories to the Search Paths

(a) In a UNIX environment:	`asmsc100 -m/sctools/fftlib trans.asm`
(b) In a Windows environment:	`asmsc100 -ic:\sctools\fftlib filter.asm`

2.2.8 Defining Substitution Strings

Substitution strings may be defined on the command line using the `-d` option, as follows:

`-d` *`symbol string`*	Causes the assembler to replace every occurrence of `symbol` in the source file with the specified `string`. The string must be preceded by a space and enclosed in single quotes if it contains embedded spaces. The `-d` *`symbol string`* sequence can be repeated as often as needed on the command line.

In Example 2-8, all occurrences of `BIG_ENDIAN` in the source file, `vit.asm`, are replaced with the string, ‘1’.

Example 2-8. Defining Substitution Strings Using the -d Option

```
asmsc100 -b -d BIG_ENDIAN 1 -obe vit.asm
```

Substitution strings may also be defined directly in the source file using the DEFINE directive. See page 5-30 in Chapter 5, “Assembler Directives and Significant Characters,” for details.

2.2.9 Using OPT Options on the Command Line

Any of the options available with the OPT directive may be used on the command line when prefixed with `-o`. Multiple options must be separated by commas with no intervening spaces.

Example 2-9 illustrates two ways of specifying the `MD` and `MEX` options, which instruct the assembler to print macro definitions and macro expansions in the source listing.

Example 2-9. Two Ways of Specifying Assembler Options

(a) On the command line with `-o`:

```
asmsc100 -b -l -omd,mex corr.asm
```

(b) In the assembly source file as an argument to the OPT directive:

```
OPT MD,MEX
```

For a list of all available OPT options, see Table 5-3 on page 5-57.

2.2.10 Specifying Big-Endian Mode

The following options control whether the assembler operates in little-endian mode or big-endian mode.

`le`	Generates little-endian object files, in which the least significant byte of a word occupies the lower address. This is the default.
`be`	Generates big-endian object files, in which the most significant byte of a word occupies the lower address.

The endian mode can be specified in two ways:

- As an argument to the `-o` option on the command line
- As an argument to the OPT directive, placed at the beginning of the source file

Example 2-10 illustrates how to invoke the assembler in big-endian mode, using the `-obe` command-line option.

Example 2-10. Invoking the Assembler in Big-Endian Mode

```
asmsc100 -b -l -obe vit.asm
```

2.2.11 Enabling Restriction Checking

Error messages associated with SC100 assembly programming restrictions are listed in Table 2-2. Each message is identified by a tag number, which corresponds to a restriction with a similar identifier in the *SC140 DSP Core Reference Manual*. For example, the `a1` and `gg4` tags listed in Table 2-2 correspond to Restriction A.1 and Restriction G.G.4, respectively, in the Core Reference Manual.

Currently, the assembler does not perform error checks for these restrictions unless specifically instructed to do so. Restriction checking can be requested using the `-s` option, as follows:

`-s` *`tag`*[,*`tag`*] Enable error checking for one or more individual restrictions by listing the tag number associated with the restriction. A space must separate the `-s` option and the first `tag` argument. Multiple tags must be separated by commas with no intervening spaces. The `tag` arguments are not case sensitive.

The following command instructs the assembler to check `myprog.asm` for violations of the A.1, A.2, and G.G.1 restrictions:

```
asmsc100 -b -s a1,a2,gg1 myprog.asm
```

`-s all` Enable error checking for all restrictions by specifying `-s all` on the command line, as follows:

```
asmsc100 -b -s all myprog.asm
```

Table 2-2 lists the SC140 restriction error messages.

Table 2-2. Restriction Error Messages

Tag	Error Message
`a1`	At least two cycles are required between MCTL modification and address pointer usage MCTL register write and R register usage not permitted within a group
`a2`	AGU register contents are not available for an additional cycle AGU register contents may be affected by register write in previous cycle
`a3`	A valid group must follow a group containing JT, JF, or TRAP
`cl1`	Parallel PCTL access not allowed in a group
`cl2`	At least 7 cycles required between PCTL writes or STOP/WAIT
`cl3`	PCTL write in a group with STOP/WAIT not allowed
`d1`	Illegal instruction in delay slot
`d2`	Instruction which uses the SR is not allowed in RTED delay slot
`d3`	RTE/D and instruction which uses SR not permitted in same group
`d4`	Instruction is not allowed in RTSD/RTED/RTSTKD delay slot
`d5`	Instruction is not allowed in JSRD delay slot JSR/D and moves to SR are not permitted in the same group
`gg1`	Too many total instructions
`gg2`	Total instruction length cannot exceed eight words (including prefix)
`gg3`	Too many total DALU ops plus BFU ops Only one bit mask instruction permitted in group
`gg4`	Only one T bit update instruction permitted in group Only one SP update instruction permitted in group Only one address register update permitted in group Double push may only include one register from eeeee or EEEEE Double pop may only include one register from eeeee or EEEEE Only one change of flow permitted in group

Table 2-2. Restriction Error Messages (Continued)

Tag	Error Message
`gg5`	DALU register may only be used four times per set
`gp1`	Too many extension words in paired grouping
`gp3`	Only one di or ei permitted in group Only one stop or wait permitted in group Only one debug instruction permitted in group Only one mark instruction permitted in group
`gp3a`	DOEN type instruction may not be grouped with break
`gp4`	RTE/D and AGU or IFT/F/A combination not permitted in same group
`gp5`	N or M used as source register more than once in a group
`gs`	Pairing of non-pairable instructions
`lc1`	Branch to last two execution sets in a loop not allowed
`lc2`	Change of flow not allowed between LA and LA-2
`lc3`	No conditional branch is allowed in the execution set before the start address of a short loop
`lc5`	No conditional branches allowed in the last four execution sets of a long loop
`lc7`	Destination address of SKIP/BREAK/CONT cannot be in the same loop
`lc9`	SKIPS/BREAK/CONT target cannot be followed by consecutive loop addresses
`lc10`	JSR/BSR to LA-2 of long loop or SA of short loop
`ld1`	SKIPLS instruction not allowed immediately after DOEN/SH or move to LC
`ld2`	Three, four, or more execution sets are required between DOEN or write to LC and loop end
`ld3`	Two, three, or more execution sets are required between DOENSH or write to LC and the first execution set
`ld4`	Write to SR is not allowed before DOEN/SH
`ld5`	One, two or more execution sets required between LC update and CONT/D instruction
`ld6`	At least three execution sets required between DOSETUP and last execution set
`ld7`	At least one execution set required between CONT/D and modification of SA
`ld8`	At least three execution sets required between read of LC and last set of loop
`ld9`	At least one execution set required between read of LC and first set of short loop
`lg3`	Less than three sets between MOVE/PUSH SR and end of loop
`lg4`	Short loop SA follows MOVE/PUSH SR
`ll1`	Illegal delay slot instruction in last two execution sets of a loop
`ll2`	DOEN/SH or write to LC not allowed in last three execution sets of a loop
`ll3`	Illegal delay slot instruction in short loop
`ln1`	Nested loops cannot end in same address
`ln2`	A loop may only be nested inside a loop of a smaller DOEN number
`ln3`	Consecutive DOEN without intervening loopstart
`ln4`	Nested short loopend cannot occur in last two execution sets of outer loop
`r2`	Cannot compare a register to itself

Table 2-2. Restriction Error Messages (Continued)

Tag	Error Message
`sr2`	Instruction not allowed within two execution sets of an SR change Instruction may be affected by previous SR change
`sr3`	Instruction not allowed after SR change
`t1`	IFc cannot follow a group containing a T bit modification
`v1`	Offset + Width > 40 bits
`v2`	Offset + Width > 40 bits
`v3`	SP must be a multiple of eight

2.3 Assembler Processing

The SC100 Assembler is a three-pass assembler. Figure 2-1 highlights the assembler operations in each pass.

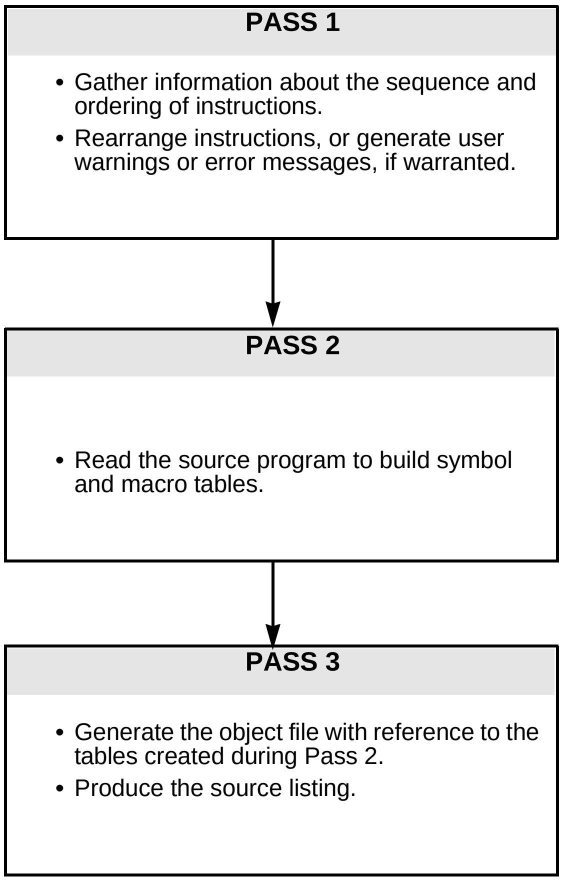

PASS 1
• Gather information about the sequence and ordering of instructions.
• Rearrange instructions, or generate user warnings or error messages, if warranted.
PASS 2
• Read the source program to build symbol and macro tables.
PASS 3
• Generate the object file with reference to the tables created during Pass 2.
• Produce the source listing.

Figure 2-1. Assembler Operation

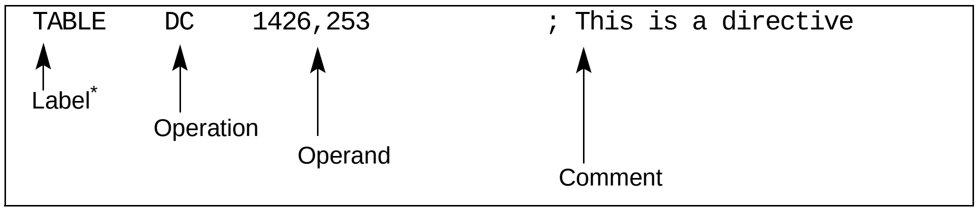
The assembler processes each source statement completely before it reads the next source statement. As it reads each line, the assembler applies translations specified by all DEFINE directives. It then examines the label, operation code, and operand fields. The assembler scans the macro definition table for a match with the operation code. If no match is found, the assembler scans the operation code and directive tables for a match with a known opcode.

The assembler displays an error before printing the actual line containing the error. The assembler accumulates errors and warnings, and prints the totals at the end of the source listing. Error messages are always displayed regardless of whether the source listing is generated. The number of errors is returned as an exit status when the assembler returns control to the host operating system.

2.4 Source Statements

Assembly language programs consist of a sequence of source statements that contain one or more assembly language instructions and a comment field, or an assembler directive and a comment field.

As shown in Figure 2-2, a source statement may include four fields in its most basic form: label, operation, operand, and comment. These fields are described in the following sections.



*Begins in column 1

Figure 2-2. Basic Source Statement

The following rules apply to source statements:

- Anything in the first column is considered a label.
- Only fields preceding the comment field are significant to the assembler; the comment field is ignored.
- Fields other than the comment field cannot contain embedded whitespace characters, since these characters are used as field delimiters. An exception is spaces and tabs in quoted strings.
- A source statement can be extended to multiple lines by including the line continuation character (\) as the *last* character on the line to be continued.

  An exception to this is instruction groups, which can span multiple lines as long as the instruction group is surrounded by brackets ([ ]).
- A source statement (first line and any continuation lines) can be a maximum of 4000 characters long.
- Upper and lower case letters are equivalent for assembler mnemonics and directives, but are distinct for labels, symbols, directive arguments, and literal strings.
- If the source file contains horizontal tabs (ASCII $09), the assembler expands these to the next fixed tab stop located at eight-character intervals (column 1, 9, 17...), unless they are reset using the TAB directive. This is only significant if tab characters are embedded within literal strings.

2.4.1 Label Field

Labels begin in the first column of a source statement. A space or tab in the first column ordinarily indicates that the label field is empty. Labels are subject to the following rules:

- Labels may be indented if the label symbol is immediately followed by a colon (:) with no intervening spaces. In this case, all characters preceding the label on the line must be spaces or tabs.
- A line consisting only of a label is valid and has the effect of assigning the value of the location counter to the label. With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled.
- The first character of a label must be an alphabetic character.
- A label may occur only once in the label field of an individual source file unless it is used as a local label (discussed in Section 2.4.7, "Symbols Used as Labels") or used with the SET directive. If a non-local label occurs more than once in a label field, each reference to that label after the first is flagged as an error.

2.4.2 Operation Field

The operation field appears after the label field and must be preceded by at least one space or tab. Entries in the operation field may be one of the following types:

Opcode	Mnemonics that correspond directly to machine instructions
Directive	Special operation codes known to the assembler which control the assembly process
Macro Call	Invocation of a previously defined macro which is to be inserted in place of the macro call

The assembler first searches for operation codes in an internal macro definition table. If it does not find a match, it searches the table of machine operation codes and assembler directives. If neither of the tables holds the specified operation code, the assembler generates an error message.

This sequence can be altered using the MACLIB directive. Macro names can therefore replace standard machine operation codes and assembler directives, although a warning will be issued if such a replacement occurs.

See Section 5.2, "Assembler Directives," for more information on the MACLIB directive.

2.4.3 Operand Field

The interpretation of the operand field is dependent on the contents of the operation field. The operand field, if present, must follow the operation field and must be preceded by at least one space or tab. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas with no intervening spaces.

The operand field of machine instructions specifies the addressing mode of the instruction, as well as its operand. Addressing modes for SC140 instructions are defined in the *SC140 DSP Core Reference Manual*.

The operand field for each assembler directive is defined in Section 5.2, "Assembler Directives."

2.4.4 Comment Field

Comments are ignored by the assembler, but can be included in the source file for documentation purposes. A comment field is composed of any characters (not part of a literal string) preceded by a semicolon (;).

A comment starting in the first column of the source file will be aligned with the label field in the source listing. Otherwise, the comment will be shifted right and aligned with the comment field in the source listing, by default. Comments preceded by two consecutive semicolons (;;) are not reproduced on the source listing and are not saved as part of a macro definition.

2.4.5 Instruction Groups

Instruction groupings, called execution sets, allow multiple instructions to be executed in parallel. Following are general restrictions that apply to instruction groups. For a complete discussion of grouping restrictions, refer to the *SC140 DSP Core Reference Manual*.

- Each SC140 instruction group can contain up to six instructions: four DALU operations and two AGU operations.
- A bit-mask operation counts as an AGU operation, and cannot be grouped with another bit-mask operation.
- The total instruction word count in an SC140 set cannot exceed eight words (including the prefix).
- Instructions that modify the same register cannot be grouped together. Following are two important examples of this restriction:
  - Two change-of-flow instructions can be regarded as dual PC updates, and are therefore restricted.
  - Only one instruction can update the T (true) bit in the status register.

The assembler interprets each line containing instructions as an instruction group. This is illustrated in Example 2-11.

Example 2-11. Single-Line Instruction Group

```
     move.f (r2)+,d0     move.f (r2)+,d8     clr d5     ; An execution set with 3 instructions
```

When delimited with brackets ([ ]), an instruction group may span multiple lines, as illustrated in Example 2-12. Note the following about this example:

- No more than two instructions and one comment are included on each line. This is to facilitate readability; it is not a requirement.
- DALU and AGU instructions are separated, beginning with DALU instructions and ending with AGU instructions.
- The instruction group is delimited with brackets.

Example 2-12. Multiple-Line SC140 Instruction Group

```
        [
          mac d0,d1,d2            mac d3,d4,d5            ; multiply operands
          add d0,d1,d3            add d3,d4,d6            ; add operands
          move.f (r0)+,d0         move.w (r1)+,d1         ; load new operands
        ]
```

2.4.6 Symbol Names

Symbol names can be from one to 4000 characters long. The first character of a symbol must be alphabetic, either upper or lower case; any remaining characters can be either alphanumeric (A-Z, a-z, 0-9) or the underscore character (_). Upper and lower case letters in symbols are considered distinct unless the `-oIC` option is used.

a) Valid Symbol Names	b) Invalid Symbol Names
loop_1	1_loop
ENTRY	loop.e
a_B_c	

Symbol names and other identifiers beginning with a period (.) are legal, but are reserved for the system. Names of SC140 DSP core processor registers are also reserved by the assembler and cannot be used.

2.4.7 Symbols Used as Labels

Symbols may be used as labels. Labels preceded by a percent character (**%**) are viewed as local labels, and have a limited scope bounded by any two non-local labels. The local label can be referred to or defined only in source statements that are between two source lines containing non-local labels.

Use of local labels in macros represents a special case. By default, all local labels within a macro are considered distinct for the currently active level of macro expansion. These local labels are valid for the entire macro expansion and are not considered bounded by non-local labels. Therefore, all local labels within a macro must be unique. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded.

Non-local labels within a macro expansion are considered to be normal labels and therefore cannot occur more than once unless used with the SET directive. (See Section 5.2, “Assembler Directives.”)

2.4.8 Strings

One or more ASCII characters enclosed by single quotes (') constitute a literal ASCII string. In order to specify an apostrophe within a literal string, two consecutive apostrophes must appear where the single apostrophe is intended. Strings are used as operands for some assembler directives and can also be used to a limited extent in expressions.

A string may also be enclosed in double quotes (") in which case any DEFINE directive symbols contained in the string would be expanded. The double quote should be used with care inside macros since it is used as a dummy argument string operator. In that case, the macro concatenation operator can be used to escape the double-quoted string if desired.

Two strings separated by the string concatenation operator (++) are recognized by the assembler as equivalent to the concatenation of the two strings. For example, these two strings are equivalent:
'ABC'++'DEF' = 'ABCDEF'

The assembler has a substring extraction capability using brackets ([ ]). For example:
['abcdefg',1,3] = 'bcd'

Substrings may be used wherever strings are valid and can be nested. There are also functions for determining the length of a string and the position of one string within another. See Chapter 3, “Expressions,” for more information on string functions.

2.5 Source Listing

The source listing contains the original source statements, formatted for easier reading, and other assembler-generated information. Most lines in the listing correspond directly to a source statement. Lines which do not correspond directly to source statements include page headings, error messages, and expansions of macro calls or directives such as DC.

By default, the assembler generates a source listing which is routed to the standard output. Optionally, the source listing can be:

- Redirected to an arbitrary destination, such as a printer, file, or null device by using the I/O redirection facilities of the host operating system.
- Redirected to a file named as the argument to the `-l` command-line option. If no argument is present with the `-l` option, the assembler creates a source listing with the same name as the first source file encountered on the command line and adds a `.lst` extension.
- Inhibited by using the `IL` (inhibit listing) option.

2.5.1 Source Listing Example

An example source listing is shown in Figure 2-3. The areas of interest noted in Figure 2-3 are described below.

Banner	The first line of every source listing page contains the banner, which identifies the assembler and its version number, the date and time of assembly, the source file name, and the listing page number.
Titles	Lines 2 and 3 are reserved for the title and subtitle, which are defined with the TITLE and STITLE directives, respectively. When defined, the title and/or subtitle appears on each page following the first.
Field 1	The first field contains the source listing line number.
Field 2	The second field is the macro definition/expansion column. This column may contain one of the following indicators:

Indicator	Meaning
`m`	A macro definition is in progress. The lines marked with an ‘`m`’ are not assembled, but are retained for macro expansion.
`+`	A macro expansion is in progress.
`d`	A data expansion is occurring, as requested by the `-oCEX` option.
`i`	The line was skipped as a result of an IF-THEN-ELSE directive sequence.
`p`	The line was automatically generated by the assembler, to introduce a pipeline delay.

Field 3 This field contains the memory space value.

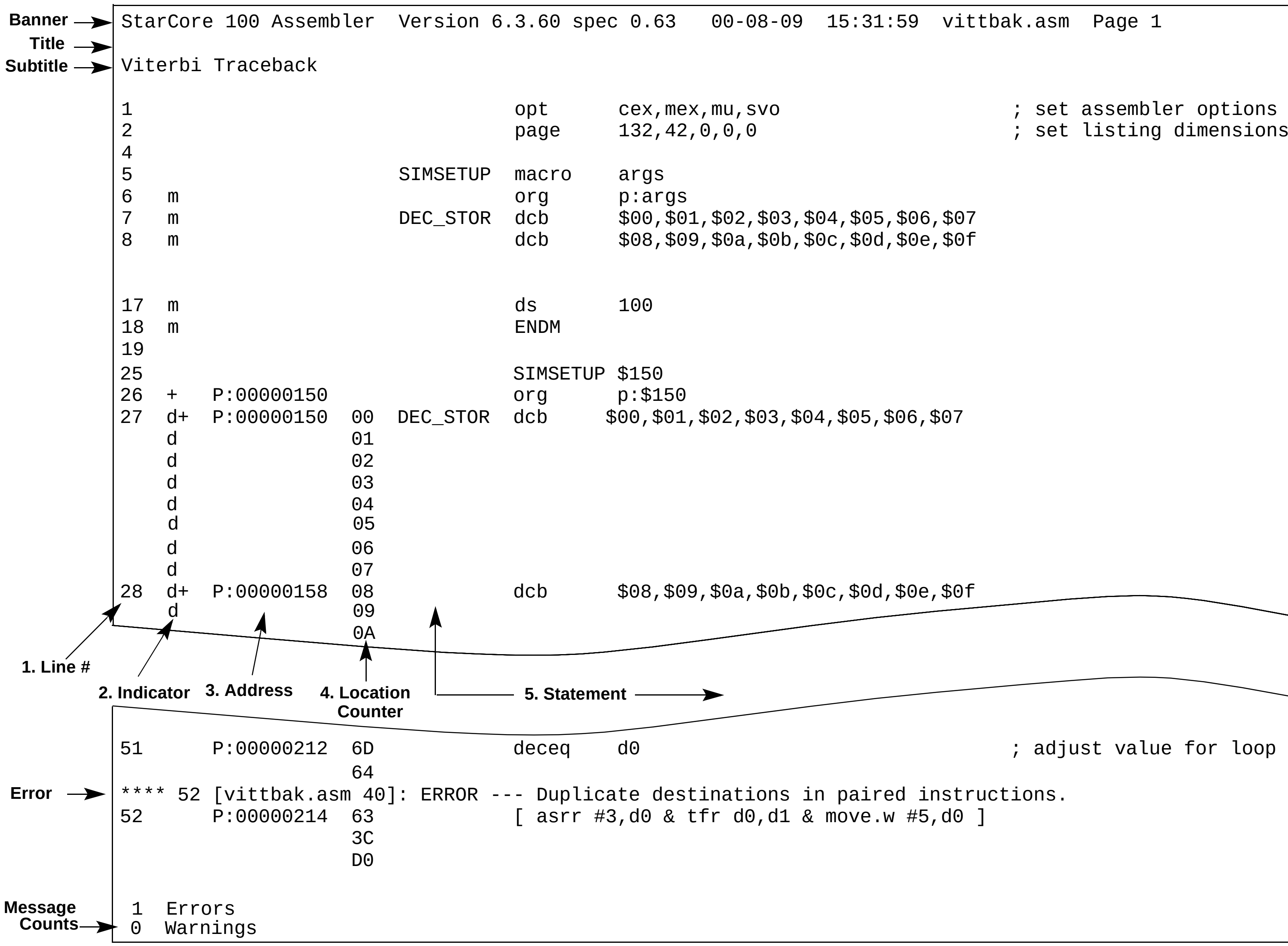
Field 4 This field contains the location counter value.

Fields 5-*n* Fields 5 and beyond contain the source statement, which may contain one or more instructions depending on whether instruction groups are used.

Error Error messages are printed in the listing above the line where the error occurred. The message notes the source file name and the source line number where the error occurred, a severity level (warning, error, or fatal), and the message text. Additional information indicating erroneous symbols or the field (Label, Opcode, Operand) where the error occurred may also be included.

Message Counts

The source listing ends with a count of the assembler errors and warnings encountered during assembly.



```
StarCore 100 Assembler  Version 6.3.60 spec 0.63   00-08-09  15:31:59  vittbak.asm  Page 1

Viterbi Traceback

1                                  opt      cex,mex,mu,svo                    ; set assembler options
2                                  page     132,42,0,0,0                      ; set listing dimensions
4
5                        SIMSETUP  macro    args
6   m                              org      p:args
7   m                    DEC_STOR  dcb      $00,$01,$02,$03,$04,$05,$06,$07
8   m                              dcb      $08,$09,$0a,$0b,$0c,$0d,$0e,$0f


17  m                              ds       100
18  m                              ENDM
19
25                                 SIMSETUP $150
26  +   P:00000150                 org      p:$150
27  d+  P:00000150  00   DEC_STOR  dcb     $00,$01,$02,$03,$04,$05,$06,$07
    d               01
    d               02
    d               03
    d               04
    d               05
    d               06
    d               07
28  d+  P:00000158  08             dcb      $08,$09,$0a,$0b,$0c,$0d,$0e,$0f
    d               09
                    0A

51      P:00000212  6D             deceq    d0                                ; adjust value for loop
                    64
**** 52 [vittbak.asm 40]: ERROR --- Duplicate destinations in paired instructions.
52      P:00000214  63             [ asrr #3,d0 & tfr d0,d1 & move.w #5,d0 ]
                    3C
                    D0


 1  Errors
 0  Warnings
```

Figure 2-3.   Assembler Source Listing

2.5.2 Memory Utilization Report

When requested, the assembler generates a memory utilization report. This is a memory map showing data allocation, code generation, and unused memory areas, along with associated label and section information, if available.

The memory utilization report is requested using the `MU` option, and is printed at the end of the source listing.

An example of a memory utilization report is shown in Figure 2-4. For each reported block, the starting and ending addresses, length, type, and any label or section data are output.

The blocks are defined in the source file by a data allocation directive (BSC, DC, DS, DSR) or by an ORG directive. See Section 5.2, “Assembler Directives,” for a description of these directives.

```
StarCore 100 Assembler Version 6.3.60 spec 0.63 00-08-09 15:31:59 vittbak.asm  Page 7
Viterbi Traceback


                            Memory Utilization Report

P Memory

Start      End          Length      Type     Label        Section     Overlay Address
00000000    0000014F          336   UNUSED
00000150    0000019F           80   CONST    DEC_STOR
000001A0    00000203          100   DATA
00000204    00000231           46   CODE
00000232    FFFFFFFF   4294966734   UNUSED
```

Figure 2-4. Optional Source Listing Information

Chapter 3
Expressions

An expression is a combination of symbols, constants, operators, and parentheses which represent a value that is used as an operand of an assembler instruction or directive. Expressions may contain user-defined labels and their associated integer or floating-point values, and/or any combination of integers, floating-point numbers, or ASCII literal strings. In general, white space (a blank or tab) is not allowed between the terms and operators of an assembler expression. Expressions otherwise follow the conventional rules of algebra and boolean arithmetic.

3.1 Absolute and Relative Expressions

An expression may be either *relative* or *absolute*. An absolute expression is one which consists only of absolute terms, or is the result of two relative terms with opposing signs. A relative expression consists of a relative term by itself or only in combination with absolute terms.

When the assembler operates in relative mode, all address expressions must adhere to the above definitions for absolute or relative expressions. This is because only these types of expressions retain a meaningful value after program relocation. For example, when relative terms are paired with opposing signs, the result is the difference between the two relative terms, which is an absolute value. However, if two positive relative terms are added together, the result is unpredictable based on the computed values of the terms at relocation time.

3.2 Expression Memory Space Attributes

A symbol is associated with either an integer or a floating-point value which is used in place of the symbol during the expression evaluation. Each symbol carries a memory space attribute of either P (program) or N (none).

The result of an expression always has a memory space attribute associated with it. Labels, constants, and floating-point expressions associated with the SET directive always have a memory space attribute of N. The unary logical negate operator, relational operators, logical operators, and some functions return values that have a memory space attribute of N. The result of an expression that has only one operand (and possibly the unary negate or unary minus operator) always has the memory attribute of that operand. Expressions involving operands with different memory space attributes generate a result where the memory space attribute is P.

The memory space attribute is regarded by the assembler as a type, in the same sense that high-level languages use type for variables. Symbols that are assigned memory space attributes of P are assumed to be addresses and therefore can only have values between zero and the maximum address value of the DSP inclusive. Only symbols that have a memory space attribute of N can have values greater than the maximum address of the target processor.

The memory space is implicitly P when an address is used as the operand of a LOOP, branch, or jump-type instruction.

Expressions used for immediate addressing can have any memory space attribute.

3.3 Internal Expression Representation

Expression value representation internal to the assembler is dependent on the word size of the target processor. The assembler supports a word and a double word integer format internally. The actual storage size of an expression value is dependent upon the magnitude of the result, but the assembler is capable of representing signed integers up to 64 bits in length.

Internal floating-point representation is almost entirely dependent upon the host environment, but in general, floating-point values are stored in double precision format. This means that there are ordinarily 64 bits of storage allotted for a floating-point number by the assembler, with 11 bits of exponent, 53 bits of mantissa, and an implied binary point.

3.4 Constants

Constants represent quantities of data that do not vary in value during the execution of a program.

3.4.1 Numeric Constants

Numeric constants can be in one of the bases described in Table 3-1.

Table 3-1. Numeric Constants

Base	Description	Example
Binary	Consists of a percent sign (%) followed by a string of binary digits (0,1)	%11010
Hexadecimal	Consists of a dollar sign ($) followed by a string of hexadecimal digits (0-9, A-F, a-f)	$12FF $12ff
Decimal, integer	Consists of a string of decimal digits (0-9) optionally preceded by a grave accent (`)	12345
Decimal, floating point	Includes a decimal point, or an upper or lower case ‘E’ followed by the exponent	6E10 .6 2.7e2

A constant may be written without a leading radix indicator if the input radix is changed using the RADIX directive. For example, a hexadecimal constant may be written without the leading dollar sign ($) if the input radix is set to 16 (assuming an initial radix of 10). The default radix is 10. See Section 5.2, “Assembler Directives,” for more information on the RADIX directive.

3.4.2 String Constants

String constants that are used in expressions are converted to a concatenated sequence of ASCII bytes that are right aligned. Strings used in expressions are limited to the long word size of the target processor; subsequent characters in the string are ignored. Null strings have a value of 0. For example:

'ABCD'	($41424344)
'''79'	($00273739)
'A'	($00000041)
''	($00000000) ← null string
'abcdef'	($61626364)
'abc'++'de'	($61626364)

String constants greater than the maximum number of characters can be used in expressions, but the assembler truncates the value and uses only those characters that fit in a DSP long word. In this case, a warning is printed. This restriction also applies to string constants using the string concatenation operator. Handling of string constants by the DC and DCB directives is an exception to this rule; see Section 5.2, “Assembler Directives,” for details.

3.5 Operators

Most assembler operators can be used with both floating-point and integer values. The results are determined as follows:

- If one operand of the operator has a floating-point value and the other operand has an integer value, the integer is converted to a floating-point value before the operator is applied, and the result is a floating-point value.
- If both operands are integers, the result is an integer value.
- If both operands are floating point, the result is a floating-point value.

The assembler operators are described in Table 3-2. All operators can be used with both floating-point and integer values except where otherwise noted.

Table 3-2. Assembler Operators

Type	Operator	Name	Description
Unary	+	plus	Returns the value of its operand.
	-	minus	Returns the negative of its operand.
	~	one’s complement	Integer only. Returns the one’s complement of its operand. It cannot be used with a floating-point operand.
	!	logical negate	Returns an integer 1 if the value of its operand is 0; otherwise, it returns a 0. The memory space attribute of the result will be N. For example, if the symbol BUF has a value of 0, then !BUF would have a value of 1. If BUF has a value of 1000, !BUF would have a value of 0.

Table 3-2. Assembler Operators (Continued)

Type	Operator	Name	Description
Arithmetic	+	addition	Yields the sum of its operands.
	-	subtraction	Yields the difference of its operands.
	*	multiplication	Yields the product of its operand.
	/	division	Yields the quotient of the division of the first operand by the second. For integer operands, the divide operation produces a truncated integer result.
	%	mod	Used with integers, this operator yields the remainder from the division of the first operand by the second. Used with floating-point operands, this operator applies the following rules: Y % Z = Y if Z = 0 = X if Z <> 0 where X has the same sign as Y, is less than Z, and satisfies the relationship: Y = integer * Z + X
Shift	<<	shift left	Integer only. Causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.
	>>	shift right	Integer only. Causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.
Relational[1]	<	Less than	If the indicated condition is: • True, the result of the expression is an integer 1. • False, the result of the expression is an integer 0. In either case, the memory space attribute of the result is N. For example, if D has a value of 3 and E has a value of 5, then the result of the expression D<E is 1, and the result of the expression D>E is 0. Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results.
	<=	Less than or equal	
	>	Greater than	
	>=	Greater than or equal	
	==	Equal	
	!=	Not equal	
Bitwise	&	AND	Integer only. Yields the bitwise AND function of its operand.
	|	OR	Integer only. Yields the bitwise OR function of its operand.
	^	exclusive OR	Integer only. Yields the bitwise exclusive OR function of its operands.
Logical[1]	&&	Logical AND	Returns an integer 1 if both of its operands are nonzero; otherwise, it returns an integer 0.
	||	Logical OR	Returns an integer 1 if either of its operands is nonzero; otherwise, it returns an integer 0.

1. These operators are intended primarily for use with the conditional assembly IF directive, but can be used in any expression.

3.6 Operator Precedence

Expressions are evaluated with the following operator precedence:

1. Parenthetical expression (innermost first)
2. Unary plus, unary minus, one’s complement, logical negation
3. Multiplication, division, mod
4. Addition, subtraction
5. Shift
6. Relational operators: less, less or equal, greater, greater or equal
7. Relational operators: equal, not equal
8. Bitwise AND, OR, EOR
9. Logical AND, OR

Operators of the same precedence are evaluated left to right. Valid operands include numeric constants, literal ASCII strings, and symbols. The one’s complement, shift, and bitwise operators cannot be applied to floating-point operands. That is, if the evaluation of an expression (after operator precedence has been applied) results in a floating-point number on either side of any of these operators, an error will be generated.

3.7 Functions

The assembler has several built-in functions to support data conversion, string comparison, and transcendental math computations. Functions may be used as terms in any arbitrary expression, and may have zero or more arguments. The following rules apply:

- Functions must always be followed by open and closed parentheses.
- Arguments which are expressions must be absolute expressions except where noted.
- Arguments containing external references are not allowed.
- Intervening spaces are not allowed between the function name and the opening parenthesis, or between comma-separated arguments.

Assembler functions can be grouped into the following types:

- *Mathematical functions*—comprise transcendental, random value, and min/max functions.
- *Conversion function*s—provide conversion between integer, floating-point, and fixed-point fractional values.
- *String functions*—compare strings, return the length of a string, and return the position of a substring within a string.
- *Macro functions*—return information about macros.
- *Assembler mode function*s—relate to assembler operation.

Table 3-3 summarizes the functions within each type.

Table 3-3. Summary of Assembler Functions

Type	Function	Description
Mathematical	ABS	Absolute value
	ACS	Arc cosine
	ASN	Arc sine
	AT2	Arc tangent
	ATN	Arc tangent
	CEL	Ceiling function
	COH	Hyperbolic cosine
	COS	Cosine
	FLR	Floor function
	L10	Log base 10
	LOG	Natural logarithm
	MAX	Maximum value
	MIN	Minimum value
	POW	Raise to a power
	RND	Random value
	SGN	Return sign
	SIN	Sine
	SNH	Hyperbolic sine
	SQT	Square root
	TAN	Tangent
	TNH	Hyperbolic tangent
	XPN	Exponential function
Conversion	CVF	Convert integer to floating point
	CVI	Convert floating point to integer
	CVS	Convert memory space
	FLD	Shift and mask operation
	FRC	Convert floating point to fractional
	LFR	Convert floating point to long fractional
	LNG	Concatenate to double word
	LUN	Convert long fractional to floating point
	RVB	Reverse bits in field
	UNF	Convert fractional to floating point
String	LEN	Length of string
	POS	Position of substring in string
	SCP	Compare strings
Macro	ARG	Macro argument function
	CNT	Macro argument count
	MAC	Macro definition function
	MXP	Macro expansion function

Table 3-3. Summary of Assembler Functions (Continued)

Type	Function	Description
Assembler Mode	CCC	Cumulative cycle count
	CHK	Current instruction/data checksum
	CTR	Location counter type
	DEF	Symbol definition function
	EXP	Expression check
	INT	Integer check
	LCV	Location counter value
	LST	LIST directive flag value
	MSP	Memory space
	REL	Relative mode function

3.7.1 Function Descriptions

Descriptions of the assembler functions follow, each containing usage information and an example. The functions are shown as upper case in this section in order to highlight them; however, the functions are not case sensitive.

`@ABS(`*`expr`*`)` Returns the absolute value of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
MOVE #@ABS(VAL),D4.S          ; Load absolute value
```

`@ACS(`*`expr`*`)` Returns the arc cosine of `expr` as a floating-point value in the range zero to pi. The result of `expr` must be between -1 and 1. The memory space attribute of the result will be N.

Example:
```
ACOS = @ACS(-1.0)             ; ACOS = 3.141593
```

`@ARG(``{`*`symbol`*` | `*`expr`*`}`**`)`**

Returns integer 1 if the macro argument represented by `symbol` or `expr` is present, 0 otherwise. If the argument is a symbol, it must be single-quoted and refer to a dummy argument name. If the argument is an `expr`, it refers to the ordinal position of the argument in the macro dummy argument list. A warning is issued if this function is used when no macro expansion is active. The memory space attribute of the result will be N.

Example:
```
IF    @ARG(TWIDDLE)           ; Is twiddle factor provided?
```

`@ASN(`*`expr`*`)` Returns the arc sine of `expr` as a floating-point value in the range -pi/2 to pi/2. The result of `expr` must be between -1 and 1. The memory space attribute of the result will be N.

Example:
```
ARCSINE SET @ASN(-1.0)        ; ARCSINE = -1.570796
```

@AT2(*expr1*,*expr2***)**
Returns the arc tangent of `expr1`/`expr2` as a floating-point value in the range -pi to pi. Expr1 and expr2 must be separated by a comma. The memory space attribute of the result will be N.

Example:
```
ATAN EQU @AT2(-1.0,1.0)       ; ATAN = -0.7853982
```

@ATN(*expr***)** Returns the arc tangent of `expr` as a floating-point value in the range -pi/2 to pi/2. The memory space attribute of the result will be N.

Example:
```
MOVE  #@ATN(1.0),D0.S         ; Load arc tangent
```

@CCC() Returns the cumulative cycle count as an integer. Useful in conjunction with the CC, NOCC, and CONTCC assembler options (see the OPT directive on page 5-57). The memory space attribute of the result will be N.

Example:
```
IF    @CCC() > 200            ; Check if cycle count > 200
```

@CEL(*expr***)** Returns a floating-point value which represents the smallest integer greater than or equal to `expr`. The memory space attribute of the result will be N.

Example:
```
CEIL SET @CEL(-1.05)          ; CEIL = -1.0
```

@CHK() Returns the current instruction/data checksum value as an integer. Useful in conjunction with the CK, NOCK, and CONTCK assembler options (see the OPT directive on page 5-57). Note that assignment of the checksum value with directives other than SET could cause phasing errors due to different generated instruction values between passes. The memory space attribute of the result will be N.

Example:
```
CHKSUM SET @CHK()             ; Reserve checksum value
```

@CNT() Returns the count of the current macro expansion arguments as an integer. A warning is issued if this function is used when no macro expansion is active. The memory space attribute of the result will be N.

Example:
```
ARGCNT SET @CNT()             ; Reserve arg count
```

@COH(*expr***)** Returns the hyperbolic cosine of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
HYCOS EQU @COH(VAL)           ; Compute hyperbolic cosine
```

@COS(*expr***)** Returns the cosine of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
DC -@COS(@CVF(COUNT)*FREQ)    ; Compute cosine value
```

`@CTR({L|R})` If L is specified as the argument, returns the counter number of the load location counter. If R is specified, returns the counter number of the runtime location counter. The counter number is returned as an integer value with a memory space of N.

Example:
```
CNUM = @CTR(R)                ; Runtime counter number
```

`@CVF(`*`expr`*`)` Converts the result of `expr` to a floating-point value. The memory space attribute of the result will be N.

Example:
```
FLOAT SET @CVF(5)             ; FLOAT = 5.0
```

`@CVI(`*`expr`*`)` Converts the result of `expr` to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating-point values are truncated). The memory space attribute of the result will be N.

Example:
```
INT SET @CVI(-1.05)           ; INT = -1
```

`@CVS({P | N},`*`expr`*`)`
Converts the memory space attribute of `expr` to that specified by the first argument; returns `expr`. The `expr` may be relative or absolute.

Example:
```
LOADDR EQU @CVS(P,TARGET)     ; Set LOADDR to P:TARGET
```

`@DEF(`*`symbol`*`)`
Returns an integer 1 if `symbol` has been defined; otherwise, it returns a 0. The memory space attribute of the result is N. The `symbol` may be any label not associated with a MACRO or SECTION directive. If `symbol` is quoted, it is looked up as a DEFINE symbol; if it is not quoted, it is looked up as an ordinary label.

Example:
```
IF    @DEF(ANGLE)             ; Assemble if ANGLE is defined
```

`@EXP(`*`expr`*`)` Returns an integer 1 if the evaluation of `expr` would result in errors. Returns 0 if the evaluation of `expr` would cause an error. The memory space attribute of the result is N. No error will be output by the assembler if `expr` contains an error. No test is made by the assembler for warnings. The `expr` may be relative or absolute.

Example:
```
IF    !@EXP(@FRC(VAL))        ; Skip on error
```

`@FLD(`*`base`*`,`*`value`*`,`*`width`*`[,`*`start`*`])`
Shift and mask `value` into `base` for `width` bits beginning at bit `start`. If `start` is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value with a memory space attribute of N.

Example:
```
SWITCH EQU @FLD(TOG,1,1,7)    ; Turn eighth bit on
```

@FLR(*expr*) Returns a floating-point value which represents the largest integer less than or equal to `expr`. The memory space attribute of the result will be N.

Example:
```
FLOOR SET @FLR(2.5)           ; FLOOR  =  2.0
```

@FRC(*expr*) This function performs scaling and convergent rounding to obtain the fractional representation of the floating-point `expr` as an integer. The memory space attribute of the result will be N.

Example:
```
FRAC EQU @FRC(FLT)+1          ; Compute saturation
```

@INT(*expr*) Returns an integer 1 if `expr` has an integer result; otherwise, it returns a 0. The `expr` may be relative or absolute. The memory space attribute of the result will be N.

Example:
```
IF    @INT(TERM)              ; Insure integer value
```

@L10(*expr*) Returns the base 10 logarithm of `expr` as a floating-point value. `expr` must be greater than zero. The memory space attribute of the result will be N.

Example:
```
LOG EQU @L10(100.0)           ; LOG = 2
```

@LCV({L | R}[,{L | H | *expr*}])

If L is specified as the first argument, returns the memory space attribute and value of the load location counter. If R is specified, returns the memory space attribute and value of the runtime location counter. The optional second argument indicates the Low, High, or numbered counter and must be separated from the first argument by a comma. If no second argument is present, the default counter (counter 0) is assumed.

The @LCV function will not work correctly if used to specify the runtime counter value of a relocatable overlay. This is because the resulting value is an overlay expression, and overlay expressions may not be used to set the runtime counter for a subsequent overlay. See the ORG directive in Section 5.2, “Assembler Directives,” for more information.

Example:
```
ADDR = @LCV(R)                ; Save runtime address
```

@LEN(*string*)

Returns the length of `string` as an integer. The memory space attribute of the result will be N.

Example:
```
SLEN SET @LEN('string')       ; SLEN = 6
```

@LFR(*expr*) This function performs scaling and convergent rounding to obtain the fractional representation of the floating-point `expr` as a long integer. The memory space attribute of the result will be N.

Example:
```
LFRAC EQU @LFR(LFLT)          ; Store binary form
```

@LNG(*expr1*,*expr2*)
Concatenates the single word `expr1` and `expr2` into a double word value such that `expr1` is the high word and `expr2` is the low word. The memory space attribute of the result will be N.

Example:
```
LWORD DC @LNG(HI,LO)          ; Build long word
```

@LOG(*expr*) Returns the natural logarithm of `expr` as a floating-point value. `expr` must be greater than zero. The memory space attribute of the result will be N.

Example:
```
LOG EQU @LOG(100.0)           ; LOG = 4.605170
```

@LST() Returns the value of the LIST directive flag as an integer, with a memory space attribute of N. When a LIST directive is encountered in the assembler source, the flag is incremented; when a NOLIST directive is encountered, the flag is decremented.

Example:
```
DUP @CVI(@ABS(@LST()))        ; List unconditionally
```

@LUN(*expr*) Converts the double-word `expr` to a floating-point value. The `expr` should represent a binary fraction. The memory space attribute of the result will be N.

Example:
```
DBLFRC EQU @LUN($3FE0000000000000)  ; DBLFRC = 0.5
```

@MAC(*symbol*)
Returns an integer 1 if `symbol` has been defined as a macro name, 0 otherwise. The memory space attribute of the result will be N.

Example:
```
IF    @MAC(DOMUL)             ; Expand macro
```

@MAX(*expr1*[,...,*exprN*])
Returns the greatest of `expr1`,...,`exprN` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
MAX DC @MAX(1.0,5.5,-3.25)    ; MAX = 5.5
```

@MIN(*expr1*[,...,*exprN*])
Returns the least of `expr1`,...,`exprN` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
MIN DC @MIN(1.0,5.5,-3.25)    ; MIN = -3.25
```

@MSP(*expr*) Returns the memory space attribute of `expr` as an integer value where N equals 0, and P space equals 4. The `expr` may be relative or absolute.

Example:
```
MEM SET @MSP(ORIGIN)          ; Save memory space
```

@MXP() Returns an integer 1 if the assembler is expanding a macro; otherwise, returns a 0. The memory space attribute of the result will be N.

Example:
```
IF    @MXP()                  ; Macro expansion active?
```

@POS(*str1*,*str2*[,*start*])
Returns the position of string `str2` in `str1` as an integer, starting at position `start`. If `start` is not given, the search begins at the beginning of `str1`. If the `start` argument is specified, it must be a positive integer and cannot exceed the length of the source string. The memory space attribute of the result will be N.

Example:
```
ID EQU @POS('Star*Core 140','Core') ; ID = 5
```

@POW(*expr1*,*expr2*)
Returns `expr1` raised to the power `expr2` as a floating-point value. `expr1` and `expr2` must be separated by a comma. The memory space attribute of the result will be N.

Example:
```
BUF EQU @CVI(@POW(2.0,3.0))   ; BUF = 8
```

@REL() Returns an integer 1 if the assembler is operating in relative mode; otherwise, returns a 0. The memory space attribute of the result will be N.

Example:
```
IF    @REL()                  ; Check if in relative mode
```

@RND() Returns a random value in the range 0.0 to 1.0. The memory space attribute of the result will be N.

Example:
```
SEED DC @RND()                ; Save initial seed value
```

@RVB(*expr1*[,*expr2*])
Reverse the bits in `expr1` delimited by the number of bits in `expr2`. If `expr2` is omitted, the field is bounded by the target word size. Both expressions must be single word integer values.

Example:
```
REV EQU @RVB(VAL)             ; Reverse all bits in value
```

@SCP(*str1*,*str2*)
Returns an integer 1 if the two strings compare; otherwise, returns a 0. The two strings must be separated by a comma. The memory space attribute of the result will be N.

Example:
```
IF    @SCP(STR,'MAIN')        ; Check if STR equals MAIN
```

@SGN(*expr*) Returns the sign of `expr` as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The memory space attribute of the result will be N. The expression may be relative or absolute.

Example:
```
IF    @SGN(INPUT)             ; Check if sign is positive
```

@SIN(*expr*) Returns the sine of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
DC  @SIN(@CVF(COUNT)*FREQ)    ; Compute sine value
```

@SNH(*expr*) Returns the hyperbolic sine of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
HSINE EQU @SNH(VAL)           ; Hyperbolic sine
```

@SQT(*expr*) Returns the square root of `expr` as a floating-point value. `expr` must be positive. The memory space attribute of the result will be N.

Example:
```
SQRT EQU @SQT(3.5)            ; SQRT = 1.870829
```

@TAN(*expr*) Returns the tangent of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
MOVE  #@TAN(1.0),D1.S         ; Load tangent
```

@TNH(*expr*) Returns the hyperbolic tangent of `expr` as a floating-point value. The memory space attribute of the result will be N.

Example:
```
HTAN = @TNH(VAL)              ; Hyperbolic tangent
```

@UNF(*expr*) Converts `expr` to a floating-point value. The `expr` should represent a binary fraction. The memory space attribute of the result will be N.

Example:
```
FRC EQU @UNF($400000)         ; FRC = 0.5
```

@XPN(*expr*) Returns the exponential function (base e raised to the power of `expr`) as a floating-point value. The memory space attribute of the result will be N.

Example:
```
EXP EQU @XPN(1.0)             ; EXP = 2.718282
```

Chapter 4 Software Project Management

Complex software projects often are divided into smaller program units. A team of programmers may write these subprograms in parallel, or they may reuse programs written for a previous development effort.

This chapter discusses the assembler directives that assist in managing these types of software projects.

4.1 Sections

The assembler provides a pair of directives, SECTION and ENDSEC, that allow a programmer to encapsulate program units, and to define relocatable blocks of code and data so that concerns about memory placement are postponed until after the assembly process.

A section is bounded by a SECTION directive and an ENDSEC directive as shown in Example 4-1. The SECTION directive defines the start of a section, giving it the name specified by `section_name`. The ENDSEC directive defines the end of the section.

Example 4-1. Section Syntax

```
SECTION section_name [GLOBAL]
  .
  .
section source statements
  .
  .
ENDSEC
```

4.1.1 Section Names

A section may be given any name that is not reserved by the system. However, the assembler recognizes the names of several default ELF sections, whose types and attributes are predefined (see Table 4-1). When these section names are used, the assembler defaults to the predefined type and flags associated with the name.

The assembler assumes that sections with other names are code (.text) sections, and sets the type and flags accordingly. If the section is not a code section, then the programmer must override the type and flags settings using the SECTYPE and SECFLAGS directives, respectively.

Table 4-1. Default ELF Sections

Section	Contents	Type	Attributes
.text	Program code	PROGBITS	ALLOC, EXECINSTR
.data	Initialized data	PROGBITS	ALLOC, WRITE
.rodata	Read-only, initialized data	PROGBITS	ALLOC
.bss	Uninitialized data	NOBITS	ALLOC, WRITE

Names of certain specialized ELF sections are reserved and should not be used. These section names are listed in Table 4-2.

Table 4-2. Reserved Section Names

.debug_abbrev	.debug_pubname	.rel.line
.debug_info	.default	.rel.text
.debug_aranges	.hash	.shstrtab
.debug_line	.line	.strtab
.debug_macro	.note	.symtab

4.1.2 Nested and Fragmented Sections

Sections can be nested to any level. When the assembler encounters a nested section, the current section is stacked and the new section is used. When the ENDSEC directive of the nested section is encountered, the assembler restores the old section and uses it. The ENDSEC directive always applies to the most recent SECTION directive.

Sections may also be split into separate parts. That is, the section name can be used multiple times with SECTION and ENDSEC directive pairs. The reuse of the section name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve P space storage locations grouped together).

4.1.3 Sections and Symbols

Symbols defined outside a section are considered global. They can satisfy an outstanding reference in the current file at assembly time, or in any file at link time. Global symbols may be referenced freely from inside or outside any section, as long as the global symbol name does not conflict with another symbol of the same name.

Symbols defined within a section are considered local, by default. Any reference to a local symbol can be satisfied in the file in which it is defined. Local symbols within sections may be declared global as follows:

- An individual symbol may be declared global by using the GLOBAL directive (see page 5-46).
- All symbols within a section (up to the next ENDSEC directive) may be declared global by using the GLOBAL qualifier to the SECTION directive (see page 5-68).

4.1.4 Macros and DEFINE Symbols within Sections

Macros and DEFINE directive symbols that are defined within a section are considered local. When defined outside of any section, these symbols are globally applied.

Macros and DEFINE symbols defined in a section can never be accessed globally. If global accessibility is desired, the macros and DEFINE symbols should be defined outside of any section.

4.2 Sections and Relocation

Sections provide the basic grouping for relocation of code and data blocks. Any code or data inside a section is considered an indivisible block with respect to relocation, and is independently relocatable within the memory space to which it is bound.

For every section defined in the source, a set of location counters is allocated for the P memory space. These counters are used to maintain offsets of data and instructions relative to the beginning of the section. At link time, sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined so that each section can be relocated as a unit.

4.3 Address Assignment

The SC100 Assembler can support absolute address assignment at assembly time or generation of relocatable program addresses which are resolved during the linking phase. The ORG directive is used to specify absolute address assignment.

The ORG directive may also be used to specify which counter (the H, L, default, or numbered counter associated with that section) will be the location counter, and to assign it initial values. The names of the counters (H, L, and default) are symbolic only; the assembler does not perform checks to insure that the value assigned to the H counter is greater than the L counter. Moreover, there is no inherent relationship among numbered counters, except that counters 0, 1, and 2 correspond to the default, L, and H counters, respectively.

Counters are useful for providing mnemonic links among individual memory blocks. Separate counters can be used to obtain blocks within a common section which are mapped to separate physical memories. The ORG directive is useful in multi-programmer projects because it provides a means for the individual programmer to specify in which segment of memory the code being written will be located without specifying an absolute address. Absolute address assignment can be deferred until the various components of the program are brought together.

For more information about the ORG directive, see page 5-62 of Chapter 5, "Assembler Directives and Significant Characters."

4.4 Multi-Programmer Environment Example

Typical multi-programmer projects are often split into tasks representing functional units. For discussion purposes, suppose a project has been divided into three tasks: i/o, filter, and main. Each task will be written by a separate programmer as a separate section. For example, when the I/O task has been written, there will be a file called `io.asm`, as shown in Example 4-2.

Example 4-2. io.asm File

```
          section i_o
          secflags alloc,write,noexecinstr
          global I_PORT
I_PORT
          .
          .
           source statements
          .
          .
          endsec
```

In the following discussion, assume that similar program structures exist for the `filter` and `main` sections, and are named `filter.asm` and `main.asm`, respectively.

The three source files can be combined, either by invoking a final assembly step to assign absolute addresses, or by assembling the modules separately and then linking.

4.4.1 Absolute Mode Implementation

To assemble the entire project source code, a fourth file called `project.asm` is created, as shown in Example 4-3.

Example 4-3. project.asm File

```
                                    ; initialize entry point for program
                                    ; counter
ORG  PL:$1000

INCLUDE 'main.asm'
INCLUDE 'io.asm'
INCLUDE 'filter.asm'
END  ENTRY
```

The file is assembled in absolute mode (`-a` option) to generate an executable ELF object file called project.eld:

```
asmsc100 -a -bproject.eld project.asm
```

When the end of the `main.asm` file is encountered, the assembler returns to the next sequential statement in the `project.asm` file, which directs the assembler to start taking input from the `io.asm` file. The `ORG` statement in the `project.asm` file directs the assembler to set the current memory space to P(rogram) and initialize the L(ow) location counter to $1000. The IO source program statements shown previously will be assembled at the next available Low Program memory space.

The file `filter.asm` will be assembled in a similar manner. The last statement of the `project.asm` file informs the assembler that this is the last logical source statement, and the starting address for the object module will be a label called ENTRY (which, for purposes of this example, is assumed to have been a label declared as a global in the section `main`.

4.4.2 Relative Mode Implementation

Using the assembler default mode, each of the source files is assembled separately as follows:

```
asmsc100 -bmain.eln main.asm
asmsc100 -bio.eln io.asm
asmsc100 -bfilter.eln filter.asm
```

For each section defined in the input files, a separate set of location counters is maintained such that all memory spaces for each section begin at relative address zero. The linker is invoked to combine the files, reading the address for each section and the program’s entry point from a linker command file called link.cmd:

```
sc100-ld -M -c link.cmd -o project.eld main.eln io.eln filter.eln
```

The linker processes the main, io, and filter files in the order listed on the command line, and outputs an executable file called project.eld.

Chapter 5 Assembler Directives and Significant Characters

This chapter describes the special characters that are significant to the assembler, as well as the directives which direct its behavior. Directives are commands that instruct the assembler to carry out an action during assembly, and are not always translated into machine language.

5.1 Assembler Significant Characters

There are several one- and two-character sequences that are significant to the assembler. Some have multiple meanings depending on the context in which they are used. These characters are listed in Table 5-1 and are described later in this chapter.

Special characters associated with expression evaluation are described in Chapter 3, "Expressions."

Table 5-1. Assembler-Significant Characters

Character	Description
;	Comment delimiter
;;	Unreported comment delimiter
\	Line continuation character or macro dummy argument concatenation operator
?	Macro value substitution operator
%	Macro hex value substitution operator
^	Macro local label override operator
"	Macro string delimiter or quoted string DEFINE expansion character
@	Function delimiter
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiter and instruction grouping operator
<	Short addressing mode force operator
>	Long addressing mode force operator
#	Immediate addressing mode operator
#<	Immediate short addressing mode force operator
#>	Immediate long addressing mode force operator
$	Hex constants indicator
'	String constants delimiter

5.2 Assembler Directives

Assembler directives can be grouped by function into the following types. Table 5-2 summarizes the directives within each type:

- Assembly control
- Symbol definition
- Data definition/storage allocation
- Listing control and options
- Macros and conditional assembly

Table 5-2. Summary of Assembler Directives

Type	Directive	Description
Assembly Control	COMMENT	Start comment lines
	DEFINE	Define substitution string
	END	End of source program
	FAIL	Programmer-generated error message
	HIMEM	Set high memory bounds
	INCLUDE	Include secondary file
	LOMEM	Set low memory bounds
	MSG	Programmer-generated message
	ORG	Initialize memory space and location counters
	RADIX	Change input radix for constants
	UNDEF	Undefine the DEFINE symbol
	WARN	Programmer-generated warning
Symbol Definition	ENDSEC	End section
	EQU	Equate symbol to a value
	GLOBAL	Global section symbol declaration
	GSET	Set global symbol to a value
	SECFLAGS	Set ELF section flags
	SECTION	Start section
	SECTYPE	Set ELF section type
	SET	Set symbol to a value
	SIZE	Set size of symbol in the ELF symbol table
	TYPE	Set symbol type in the ELF symbol table

Table 5-2. Summary of Assembler Directives (Continued)

Type	Directive	Description
Data Definition and Storage Allocation	ALIGN	Align location counter
	BADDR	Set buffer address
	BSB	Block storage bit-reverse
	BSC	Block storage of constant
	BUFFER	Start buffer
	DC	Define constant
	DCB	Define constant byte
	DCL	Define constant long
	DCW	Define constant word
	DS	Define storage
	DSR	Define reverse carry storage
	ENDBUF	End buffer
Listing Control and Options	LIST	List the assembly
	NOLIST	Stop assembly listing
	OPT	Set assembler options
	PAGE	Top of page/size page
	PRCTL	Send control string to printer
	STITLE	Initialize program subtitle
	TITLE	Initialize program title
Macros and Conditional Assembly	DUP	Duplicate sequence of source lines
	DUPA	Duplicate sequence with arguments
	DUPC	Duplicate sequence with characters
	DUPF	Duplicate sequence in loop
	ENDIF	End of conditional assembly
	ENDM	End of macro definition
	EXITM	Exit macro
	IF	Conditional assembly directive
	MACLIB	Macro library
	MACRO	Macro definition
	PMACRO	Purge macro definition

5.3 Descriptions

This section describes the assembler special characters and directives. Each description includes usage guidelines and examples. Some directives require a label field, while in other cases a label is optional. If the description of an assembler directive does not indicate a mandatory or optional label field, then a label is not allowed on the same line as the directive.

Note: The assembler directives in this chapter are shown in upper case to highlight them. However, the assembler recognizes both upper and lower case for directives.

;
Comment Delimiter Character

Description

A comment is any number of characters preceded by a semicolon (;), but not part of a literal string. Comments are not significant to the assembler, but can be used to document the source program. Comments are reproduced in the assembler source listing. Comments are normally preserved in macro definitions, but this option can be turned off using the NOCM option.

Comments can occupy an entire line, or can be placed after the last assembler-significant field in a source statement. A comment beginning in the first column of the source file is aligned with the label field in the source listing. Otherwise, the comment is shifted right and aligned with the comment field in the source listing.

Example 5-1. Comment Delimiter Character (;)

```
; This comment begins in column 1 of the source file
LOOP      JSR COMPUTE             ; This is a trailing comment
                                  ; these two comments are preceded
                                  ; by a tab in the source file
```

;; Unreported Comment Delimiter Characters

Description

Unreported comments are any number of characters preceded by two consecutive semicolons (;;) that are not part of a literal string. Unreported comments are not considered significant by the assembler, and can be included in the source statement, following the same rules as normal comments. However, unreported comments are never reproduced on the assembler source listing, and are never saved as part of macro definitions.

Example 5-2. Unreported Comment Delimiter Character (;;)

```
;; These lines will not be reproduced
;; in the source listing
```

\ Line Continuation Character or Macro Argument Concatenation Character

Line Continuation Character

The backslash character (\), if used as the *last* character on a line, tells the assembler that the source statement is continued on the following line. The continuation line is concatenated to the previous line of the source statement, and the result is processed by the assembler as if it were a single-line source statement. The maximum length of the source statement (the first line and any continuation lines) is 4000 characters.

Example 5-3. Line Continuation Character (\)

```
; This comment \
extends over \
three lines.
```

Macro Argument Concatenation

The backslash (\) is also used to concatenate a macro dummy argument with other adjacent alphanumeric characters. Dummy arguments that are to be concatenated with other characters must be preceded by the backslash (\) to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the backslash and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name.

In Example 5-4, the backslash (\) in the macro definition shown in (a) tells the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character R. The macro call is shown in (b), and the resulting expansion is shown in (c).

Example 5-4. Macro Argument Concatenation Character (\)

(a) Macro definition

```
SWAP_REG  MACRO REG1,REG2                        ; Swap REG1,REG2 using D4 as temp
          MOVE  R\REG1,D4
          MOVE  R\REG2,R\REG1
          MOVE  D4,R\REG2
          ENDM
```

(b) Macro call

```
          SWAP_REG 0,1
```

(c) Macro expansion

```
          MOVE R0,D4
          MOVE R1,R0
          MOVE D4,R1
```

?
Return Value of Symbol Character

Description

When used in macro definitions, the `?symbol` sequence is converted to an ASCII string representing the decimal value of `symbol`. The value of `symbol` must be an integer, not floating point. The question mark (?) operator may be used in association with the backslash (\) operator.

In Example 5-5, consider the macro definition shown in (a). If the source file contained the SET statements and macro call shown in (b), the sequence of events would be as follows:

1. The macro processor substitutes the characters AREG for each occurrence of REG1, and BREG for each occurrence of REG2. For discussion purposes (this would not appear on the source listing), the intermediate macro expansion would be:

   ```
   MOVE R\?AREG,D4
   MOVE R\?BREG,R\?AREG
   MOVE D4,R\?BREG
   ```

2. The macro processor then replaces ?AREG with the character 0, and ?BREG with the character 1. Again, for discussion purposes, the resulting intermediate expansion would be:

   ```
   MOVE R\0,D4
   MOVE R\1,R\0
   MOVE D4,R\1
   ```

3. Next, the macro processor applies the concatenation operator (\). The resulting expansion as it would appear in the source listing is shown in (c).

Example 5-5. Return Value of Symbol Character (?)

(a) Macro definition

```
SWAP_SYM  MACRO REG1,REG2                    ; Swap REG1,REG2 using D4 as temp
          MOVE  R\?REG1,D4
          MOVE  R\?REG2,R\?REG1
          MOVE  D4,R\?REG2
          ENDM
```

(b) Macro call

```
AREG      SET 0
BREG      SET 1
          SWAP_SYM AREG,BREG
```

(c) Macro expansion

```
          MOVE R0,D4
          MOVE R1,R0
          MOVE D4,R1
```

%
Return Hex Value of Symbol Character

Description

When used in macro definitions, the `%symbol` sequence is converted to an ASCII string representing the hexadecimal value of `symbol`. The value of `symbol` must be an integer, not floating point. The percent (%) operator may be used in association with the backslash (\) operator.

In Example 5-6, the macro definition shown in (a) generates a label consisting of the concatenation of the label prefix argument and a value that is interpreted as hexadecimal. If the macro were called as shown in (b), the sequence of events would be as follows:

1. The macro processor substitutes the characters HEX for LAB, then replaces %VAL with the character A, since A is the hexadecimal representation for the decimal integer 10.
2. The macro processor applies the concatenation operator (\).
3. The macro processes substitutes the string 'NOP' for the STMT argument. The resulting expansion as it would appear in the source listing would be that shown in (c).

Note: The percent sign is also the character used to indicate a binary constant. If a binary constant is required inside a macro it may be necessary to enclose the constant in parentheses or escape the constant by following the percent sign by a backslash (\).

Example 5-6. Return Hex Value of Symbol Character (%)

(a) Macro definition

```
GEN_LAB   MACRO  LAB,VAL,STMT
LAB\%VAL  STMT
          ENDM
```

(b) Macro call

```
NUM       SET  10
          GEN_LAB HEX,NUM,'NOP'
```

(c) Macro expansion

```
HEXA      NOP
```

^
Macro Local Label Override

Description

The circumflex (^), when used as a unary expression operator in a macro expansion, causes any local labels in its associated term to be evaluated at normal scope rather than macro scope. This means that any %labels in the expression term following the circumflex will not be searched for in the macro local label list. The operator has no effect on normal labels or outside of a macro expansion. The circumflex operator is useful for passing local labels as macro arguments to be used as referents in the macro. Note that the circumflex is also used as the binary exclusive OR operator.

In Example 5-7, the override operator causes the assembler to recognize the %LOCAL symbol outside the macro expansion and to use that value in the MOVE instruction. Without the override operator, the assembler would issue an error because %LOCAL is not defined within the body of the macro.

Example 5-7. Macro Local Label Override (^)

(a) Macro definition

```
LOAD      MACRO ADDR
          MOVE  P:^ADDR,R0
          ENDM
```

(b) Macro call

```
%LOCAL
          LOAD %LOCAL
```

"
Macro String Delimiter or Quoted String DEFINE Expansion Character

Macro String

The double quote ("), when used in macro definitions, is replaced by the macro processor with a single quote ('). The macro processor examines the characters between the double quotes for any macro arguments. The effect in the macro call is to transform any enclosed dummy arguments into literal strings.

Example 5-8. Macro String Delimiter Character (")

(a) Macro definition

```
CSTR      MACRO STRING
          DC "STRING"
          ENDM
```

(b) Macro call

```
          CSTR ABCD
```

(c) Macro expansion

```
          DC 'ABCD'
```

Quoted String DEFINE Expansion Character

A sequence of characters which matches a symbol created with a DEFINE directive is not expanded if the character sequence is contained within a quoted string. assembler strings are generally enclosed in single quotes ('). If the string is enclosed in double quotes ("), then DEFINE symbols are expanded within the string. In all other respects, use of double quotes is equivalent to that of single quotes.

Example 5-9. Quoted String DEFINE Expansion (")

(a) Macro definition

```
          DEFINE LONG 'short'
STR_MAC   MACRO STRING
          MSG 'This is a LONG STRING'
          MSG "This is a LONG STRING"
          ENDM
```

(b) Macro call

```
          STR_MAC sentence
```

(c) Macro expansion

```
          MSG  'This is a LONG STRING'
          MSG  'This is a short sentence'
```

@
Function Delimiter

Description

All assembler built-in functions start with the @ symbol.

See Section 3.7, "Functions," for descriptions of the functions supported by the assembler.

Example 5-10. Function Delimiter (@)

```
SVAL       EQU @SQT(FVAL)                                         ; Obtain square root
```

*
Location Counter Substitution

Description

When used as an operand in an expression, the asterisk (*) represents the current integer value of the location counter.

Example 5-11.   Location Counter Substitution (*)

```
         ORG P:$100
PBASE    EQU *+$20                                           ; PBASE = $120
```

++
String Concatenation Operator

Description

The string concatenation operator (++) concatenates any two strings. The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the string concatenation operator and the two strings.

Example 5-12. String Concatenation Operator (++)

```
        'ABC'++'DEF' = 'ABCDEF'
```

[ ]
Substring Delimiter

Syntax

[*string*,*offset*,*length***]**

Description

Brackets delimit a substring operation, which consists of the following:

`string`	The source string. This may be any legal string combination, including another substring.
`offset`	The substring starting position within the source string, beginning at 0. An error is issued if the offset exceeds the length of the source string.
`length`	The length of the desired substring. An error is issued if this length exceeds the length of the source string.

Example 5-13. Substring Delimiter ([ ])

```
        DEFINE ID ['abcdefg',1,3]                        ;ID = 'bcd'
```

[ ]
Instruction Grouping Operator

Syntax

```
[
  instruction ...
]
```

Description

Brackets are also used to denote instruction groups. The opening bracket cannot appear in the label field. The intervening instructions may appear on the same line separated by white space, or they may appear on subsequent lines.

Example 5-14. Instruction Grouping Operator

```
[
  move #1,D0
  add D0,D1,D2
]
```

<
Short Addressing Mode Force Operator

Description

Many DSP instructions allow a short form of addressing. If the assembler knows the absolute address on pass one, then the assembler picks the shortest form of addressing consistent with the instruction format. If the assembler does not know the absolute address on pass one (that is, the address is a forward or external reference), then the assembler picks the long form of addressing by default. If this is not desired, the short absolute form of addressing can be forced by prefixing the absolute address with the short addressing mode force operator (<).

In Example 5-15, the DATAST symbol shown in (a) is a forward reference; therefore, the assembler picks the long absolute form of addressing. Since the long absolute addressing mode causes the instruction to be two words as opposed to one word for the short absolute addressing mode, it is desirable to force the short absolute addressing mode as shown in (b).

Example 5-15. Short Addressing Mode Force Operator (<)

(a) Long absolute form of addressing chosen by default

```
          MOVE D0,P:DATAST
DATAST    EQU  P:$23
```

(b) Forcing the short absolute addressing mode using <

```
          MOVE D0,P:<DATAST
DATAST    EQU  P:$23
```

>
Long Addressing Mode Force Operator

Description

Many DSP instructions allow a long form of addressing. If the assembler knows the absolute address on pass one, then it picks the shortest form of addressing consistent with the instruction format. If this is not desired, then the long absolute form of addressing can be forced by prefixing the absolute address with the long addressing mode force operator (>).

In Example 5-16, because the DATAST symbol shown in (a) is *not* a forward reference, the assembler picks the short absolute form of addressing. If this is not desirable, then the long absolute addressing mode can be forced as shown in (b).

Example 5-16. Long Addressing Mode Force Operator (>)

(a) Short absolute form of addressing chosen by default

```
DATAST    EQU  P:$23
          MOVE D0,P:DATAST
```

(b) Forcing the long absolute addressing mode using >

```
DATAST    EQU  P:$23
          MOVE D0,P:>DATAST
```

#
Immediate Addressing Mode

Description

The pound sign (#) directs the assembler to use the immediate addressing mode.

Example 5-17. Immediate Addressing Mode (#)

```
CNST      EQU  $5
          MOVE #CNST,D0
```

#<
Immediate Short Addressing Mode Force Operator

Description

Many DSP instructions allow a short immediate form of addressing. If the assembler knows the immediate data on pass one (that is, it is not a forward or external reference), then the assembler picks the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the assembler picks the long form of immediate addressing by default. If this is not desired, then the short form of addressing can be forced using the immediate short addressing mode force operator (#<).

In Example 5-18, because the CNST symbol shown in (a) is not known to the assembler on pass one, the assembler picks the long immediate addressing form for the MOVE instruction. Since the long immediate addressing mode causes the instruction to be two words as opposed to one word for the immediate short addressing mode, it is desirable to force the immediate short addressing mode as shown in (b).

Example 5-18. Immediate Short Addressing Mode Force Operator (#<)

(a) Long immediate addressing form chosen for the MOVE instruction

```
         MOVE #CNST,D0
CNST     EQU  $5
```

(b) Forcing the immediate short addressing mode using #<

```
         MOVE #<CNST,D0
CNST     EQU  $5
```

#> Immediate Long Addressing Mode Force Operator

Description

Many DSP instructions allow a long immediate form of addressing. If the assembler knows the immediate data on pass one (that is, it is not a forward or external reference), then the assembler always picks the shortest form of immediate addressing consistent with the instruction. If this is not desired, then the long form of addressing can be forced using the immediate long addressing mode force operator (#>).

In Example 5-19, because the CNST symbol shown in (a) is known to the assembler on pass one, the assembler uses the short immediate addressing form for the MOVE instruction. If this is not desired, then the long immediate form of addressing can be forced as shown in (b).

Example 5-19. Immediate Long Addressing Mode Force Operator (#>)

(a) Short immediate addressing form chosen for the MOVE instruction

```
CNST     EQU $5
         MOVE #CNST,D0
```

(b) Forcing the long immediate addressing form using #>

```
CNST     EQU $5
         MOVE #>CNST,D0
```

ALIGN
Align Location Counter

Syntax

```
ALIGN arg
```

Description

The ALIGN directive advances the location counter so that it is aligned on an address boundary set by `arg`. The argument must be a power of two. If the location counter is already aligned on the specified boundary, it remains unchanged.

Example 5-20. ALIGN Directive

```
        ALIGN 4                                         ; Align location counter to
                                                        ; next long word boundary
```

BADDR
Set Buffer Address

Syntax

```
BADDR  R,expression
```

Description

The BADDR directive sets the location counter to the address of a reverse-carry buffer, the length of which in bytes is equal to the value specified in `expression`. The block of memory intended for the buffer is not initialized to any value.If the location counter is not zero, this directive first advances the location counter to a base address that is a multiple of 2^k, where:

$$2^k \geq \texttt{expression}$$

An error is issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the location counter is *not* advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The result of `expression` must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a reverse-carry buffer is designated and the specified expression is not a power of two, a warning will be issued.

The assembler does not allow a label with this directive.

See also: BSB, BUFFER, DSR

Example 5-21. BADDR Directive

```
        ORG P:$100
        BADDR R,24                                     ; Reverse buffer 24
```

BSB
Block Storage Bit-Reverse

Syntax

```
[label]     BSB  expression[,expression]
```

Description

The BSB directive causes the assembler to allocate and initialize a block of bytes for a reverse-carry buffer. The number of bytes in the block is given by the first expression, which must evaluate to an absolute integer. Each byte is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the location counter is not zero, this directive first advances the location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero. Also, if the first expression is not a power of two, a warning will be generated. Both expressions can have any memory space attribute.

If present, `label` is assigned the value of the location counter after a valid base address has been established.

Only one byte of object code will be shown on the listing, regardless of how large the first expression is. However, the location counter will be advanced by the number of bytes generated.

See also: BSC, DC

Example 5-22. BSB Directive

```
BUFFER      BSB  BUFSIZ                                  ; Initialize buffer to zeros
```

BSC
Block Storage of Constant

Syntax

```
[label]    BSC expression[,expression]
```

Description

The BSC directive causes the assembler to allocate and initialize a block of bytes. The number of bytes in the block is given by the first expression, which must evaluate to an absolute integer. Each byte is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed.

If the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero, an error will be generated. Both expressions can have any memory space attribute.

If present, `label` is assigned the value of the location counter at the start of the directive processing.

Only one byte of object code will be shown on the listing, regardless of how large the first expression is. However, the location counter will be advanced by the number of bytes generated.

See also: BSB, DC

Example 5-23. BSC Directive

```
UNUSED     BSC  $2FFF-@LCV(R),$FFFFFFFF                    ; Fill unused EPROM
```

BUFFER
Start Buffer

Syntax

```
BUFFER R,expression
```

Description

The BUFFER directive indicates the beginning of a reverse-carry buffer. This directive requires the following argument:

`expression`	This specifies the buffer size, in bytes. The result of `expression` must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If the expression is not a power of two, a warning is issued.

Data is allocated for the buffer until an ENDBUF directive is encountered. If less data is allocated than the size of the buffer, the remaining buffer locations are uninitialized. If more data is allocated than the specified size of the buffer, an error is issued.

Instructions and most data definition directives may appear between the BUFFER and ENDBUF pair. However, BUFFER directives may not be nested, and the following directives may not be used:

- MODE
- ORG
- SECTION
- Other buffer allocation directives

The BUFFER directive sets the location counter to the address of a buffer of the given type, the length of which in bytes is equal to the value of the specified expression. If the location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where:

$$2^k \geq expression$$

An error is issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is *not* advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The assembler does not allow a label with this directive.

See also: BADDR, BSB, DSR, ENDBUF

Example 5-24. BUFFER Directive

```
           ORG P:$100
           BUFFER R,28                                      ; Reverse buffer 28
R_BUF      DC 0.5,0.5,0.5,0.5
           DS 20                                            ; Remainder uninitialized
           ENDBUF
```

COMMENT
Start Comment Lines

SYNTAX

```
COMMENT delimiter
.
.
delimiter
```

DESCRIPTION

The COMMENT directive defines one or more lines as comments. The first non-blank character after the COMMENT directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter is considered the last line of the comment. The comment text can include any printable characters and the comment text will be reproduced in the source listing as it appears in the source file.

The assembler does not allow a label with this directive.

Example 5-25. COMMENT

```
         COMMENT  +  This is a one-line comment +
         COMMENT  *  This is a multiple-line
                     comment. Any number of lines
                     can be placed between the two delimiters.
                  *
```

DC, DCW
Define Constant

Syntax

```
[label]   DC  arg[,arg,...]
[label]   DCW arg[,arg,...]
```

Description

The DC and DCW directives allocate and initialize two bytes of memory for each argument. The components of these directives are as follows:

`label`	An optional label which, if present, is assigned the value of the location counter at the start of the directive processing.
`arg`	An argument which may be an integer constant, fractional constant, symbol, or expression. Multiple arguments must be separated by commas with no intervening spaces.

See also: DCB, DCL

Example 5-26. DCW Directive

```
                                          ; Assuming little endian mode:
      DC $1234                            ;  $1234 = $34
                                          ;          $12
      DC .15                              ;    .15 = $33
                                          ;          $13
```

DCB
Define Constant Byte

Syntax

```
[label]    DCB arg[,arg,...]
```

Description

The DCB directive allocates and initializes a byte of memory for each argument. The components of this directive are as follows:

`label`	An optional label which, if present, is assigned the value of the location counter at the start of the directive processing.
`arg`	An argument which may be a byte integer constant, a single- or multiple-character string constant, a symbol, or a byte expression. Multiple arguments must be separated by commas with no intervening spaces.

Multiple arguments are stored in successive byte locations. One or more multiple arguments can be null (indicated by two adjacent commas), in which case the corresponding byte location will be filled with zeros.

Integer arguments are stored as is, but must be byte values (for example, within the range 0-255); floating-point numbers are not allowed. Single- and multiple-character strings are handled in the following manner:

- Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

  ```
  'R' = $52
  ```
- Multiple-character strings are stored in consecutive byte addresses in the order in which they appear, as shown below. The standard C language escape characters are permitted if the `AEC` option is present.

  ```
  'AB',,'CD' =   $41
                 $42
                 $00
                 $43
                 $44
  ```

See also: DC, DCL

Example 5-27. DCB Directive

```
TABLE     DCB 'two',0,'strings',0
CHARS     DCB 'A','B','C','D'
```

DCL
Define Constant Long

Syntax

`[label]   DCL arg[,arg,...]`

Description

The DCL directive allocates and initializes four bytes of memory for each argument. The components of this directive are as follows:

`label`	An optional label which, if present, is assigned the value of the runtime location counter at the start of the directive processing.
`arg`	An argument which may be an integer constant, fractional constant, symbol, or expression. Multiple arguments must be separated by commas with no intervening spaces.

See also: DC, DCB

Example 5-28. DCL Directive

```
        DCL $12345678                               ; Assuming little endian mode,
                                                    ; $12345678 = $78
                                                    ;             $56
                                                    ;             $34
                                                    ;             $12
```

DEFINE
Define Substitution String

Syntax

```
DEFINE symbol string
```

Description

The DEFINE directive defines substitution strings that will be used on all following source lines. The assembler searches all succeeding lines for an occurrence of `symbol`, and replaces it with `string`.

The `symbol` must adhere to the restrictions for non-local labels. DEFINE symbols that are defined within a section are considered local symbols. DEFINE symbols that are defined outside a section are considered global.

Macros represent a special case. DEFINE directive translations will be applied to the macro definition as it is encountered. When the macro is expanded, any active DEFINE directive translations will again be applied.

The assembler does not allow a label with this directive.

See also: GSET, SET, UNDEF

Example 5-29. DEFINE Directive

```
          DEFINE ARRAYSIZ '16*SAMPLSIZ'
SAMPLSIZ  EQU    16
          DS     ARRAYSIZ                  ; This line transformed to
          .                                ;   DS 16*SAMPLSIZ
          .
```

DS
Define Storage

Syntax

[label] **DS** *expression*

Description

The DS directive reserves a block of bytes in memory. The reserved block of memory is not initialized to any value.

The components of the DS directive are as follows:

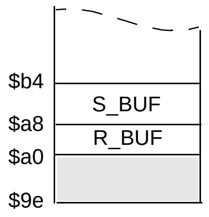
`label`	An optional label which, if present, is assigned the value of the location counter at the start of directive processing.
`expression`	This specifies the number of bytes to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

See also: DSR

Example 5-30. DS Directive

```
; Assume the current loader address is $9e

      align 16    ; Align on next 16-byte boundary
R_BUF DS    8     ; Reserve 8 bytes for R_BUF
S_BUF DS    12    ; Reserve 12 bytes for S_BUF
```



DSR
Define Reverse-Carry Storage

Syntax

[*label*] **DSR** *expression*

Description

The DSR directive reserves a block of bytes in memory for a reverse-carry buffer. The reserved block of memory is not initialized to any value.

The DSR directive advances the location counter as described below:

1. If the location counter is not zero, it is first advanced to a base address that is a multiple of 2^k and where $2^k \geq$ `expression`. An error is issued if there is insufficient memory remaining to establish a valid base address.
2. Once a valid base address has been established, the location counter is advanced a second time, by the value of `expression`.

The components of the DSR directive are as follows:

`label`	An optional label which, if present, is assigned the value of the location counter after a valid base address has been established.
`expression`	This specifies the number of bytes to be reserved. The expression must evaluate to an absolute integer greater than zero, and cannot contain any forward references (symbols that have not yet been defined). A warning is generated if `expression` is not a power of two.

See also: DS

Example 5-31. DSR Directive

```
      ORG P:$100  ; Set address to P:$100
R_BUF DSR 8       ; Reserve 8 bytes for R_BUF
```

DUP
Duplicate Sequence of Source Lines

Syntax

```
[label]    DUP expression
           .
           .
           ENDM
```

Description

The sequence of source lines between the DUP and ENDM directives will be duplicated by the number specified by the integer `expression`, which can have any memory space attribute. If the `expression` evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The DUP directive may be nested to any level.

If present, `label` is assigned the value of the runtime location counter at the start of the DUP directive processing.

See also: DUPA, DUPC, DUPF, DUPA, MACRO

Example 5-32. DUP Directive

(a) Source input statements

```
COUNT SET  3
      DUP  COUNT    ; ASR BY COUNT
      ASR  D0
      ENDM
```

(b) Resulting source listing (assumes the MD and MEX options are enabled)

```
COUNT SET  3
      DUP  COUNT    ; ASR BY COUNT
      ASR  D0
      ASR  D0
      ASR  D0
      ENDM
```

DUPA
Duplicate Sequence With Arguments

Syntax

```
[label]   DUPA  dummy,arg[,arg,...]
          .
          .
          ENDM
```

Description

The block of source statements defined by the DUPA and ENDM directives will be repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

If present, `label` is assigned the value of the runtime location counter at the start of the DUPA directive processing.

See also: DUP, DUPC, DUPF, ENDM, MACRO

Example 5-33. DUPA Directive

(a) Source input statements

```
DUPA VALUE,12,32,34
DC   VALUE
ENDM
```

(b) Resulting source listing (assumes the MD and MEX options are enabled)

```
DUPA VALUE,12,32,34
DC   12
DC   32
DC   34
ENDM
```

DUPC
Duplicate Sequence With Characters

Syntax

```
[label]     DUPC dummy,string
            .
            .
            ENDM
```

Description

The block of source statements defined by the DUPC and ENDM directives will be repeated for each character of `string`. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

If present, `label` is assigned the value of the runtime location counter at the start of the DUPC directive processing.

See also: DUP, DUPA, DUPF, ENDM, MACRO

Example 5-34. DUPC Directive

(a) Source input statements

```
DUPC VALUE,'123'
DC   VALUE
ENDM
```

(b) Resulting source listing (assumes the MD and MEX options are enabled)

```
DUPC VALUE,'123'
DC   1
DC   2
DC   3
ENDM
```

DUPF
Duplicate Sequence in Loop

Syntax

```
[label]    DUPF  dummy[,start],end[,increment]
           .
           .
           ENDM
```

Description

The block of source statements defined by the DUPF and ENDM directives will be repeated in general (`end - start`) + 1 times when `increment` is 1. The operands of this directive are as follows:

`label`	An optional label which, if present, is assigned the value of the runtime location counter at the start of the DUPF directive processing.
`dummy`	A parameter that holds the loop index value; may be used within the body of instructions.
`start`	The starting value for the loop index; defaults to 1 if omitted.
`end`	The final value.
`increment`	The increment for the loop index; defaults to 1 if omitted.

See also: DUP, DUPA, DUPC, ENDM, MACRO

Example 5-35. DUPF Directive

(a) Source input statements

```
DUPF NUM,0,7
MOVE #0,R\NUM
ENDM
```

(b) Resulting source listing (assumes the MD and MEX options are enabled)

```
DUPF NUM,0,7
MOVE #0,R0
MOVE #0,R1
MOVE #0,R2
MOVE #0,R3
MOVE #0,R4
MOVE #0,R5
MOVE #0,R6
MOVE #0,R7
ENDM
```

END
End of Source Program

Syntax

END [*expression*]

Description

The optional END directive indicates that the logical end of the source program has been encountered. Any statements following the END directive are ignored.

The optional `expression` specifies the starting execution address of the program. This argument may only be used in absolute mode and must have a memory space attribute of P (program) or N (none).

The END directive cannot be used in a macro expansion.

The assembler does not allow a label with this directive.

Example 5-36. END Directive

```
          END  BEGIN                       ; BEGIN is the starting execution address
```

ENDBUF
End Buffer

Syntax

ENDBUF

Description

The ENDBUF directive signifies the end of a buffer block. The location counter remains just beyond the end of the buffer when the ENDBUF directive is encountered.

The assembler does not allow a label with this directive.

See also: BUFFER

Example 5-37. ENDBUF Directive

```
          ORG P:$100
          BUFFER R,64                         ; Uninitialized reverse-carry buffer
          ENDBUF
```

ENDIF
End of Conditional Assembly

Syntax

```
ENDIF
```

Description

The ENDIF directive is paired with the IF directive and signifies the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the ENDIF directive always refers to the most previous IF directive.

The assembler does not allow a label with this directive.

See also: IF

Example 5-38. ENDIF Directive

```
        IF @REL()
SAVEPC  SET *                                   ; Save current program counter
        ENDIF
```

ENDM
End of Macro Definition

Syntax

ENDM

Description

The ENDM directive terminates the MACRO, DUP, DUPA, DUPC, and DUPF directives.

The assembler does not allow a label with this directive.

See also: DUP, DUPA, DUPC, DUPF, MACRO

Example 5-39. ENDM Directive

```
SWAP_SYM  MACRO REG1,REG                        ;Swap REG1,REG2 using D4.L as temp
          MOVE  R\?REG1,D4.L
          MOVE  R\?REG2,R\?REG1
          MOVE  D4.L,R\?REG2
          ENDM
```

ENDSEC
End Section

Syntax

```
ENDSEC
```

Description

The ENDSEC directive is paired with the SECTION directive and signifies the end of a section.

The assembler does not allow a label with this directive.

See also: SECTION

Example 5-40. ENDSEC Directive

```
         SECTION .data
VALUES   BSC $100                                          ; Initialize to zero
         ENDSEC
```

EQU
Equate Symbol to a Value

Syntax

label   **EQU** [**P:**]*expression*

Description

The EQU directive assigns the value of `expression` to the symbol specified by `label`. If `expression` has a memory space attribute of None, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than None and it is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The EQU directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if SECTION directives are being used). The `expression` may be relative or absolute, but cannot include a symbol that is not yet defined (no forward references are allowed).

Example 5-41. EQU Directive

```
A_D_PORT  EQU P:$4000             ; Assigns the value $4000 with a memory space
                                  ; attribute of P to the symbol A_D_PORT.

COMPUTE   EQU @LCV(L)             ; Assigns the value and memory space attribute
                                  ; of the location counter specified by
                                  ; @LCV(L) to the symbol COMPUTE.
```

EXITM
Exit Macro

Syntax

```
EXITM
```

Description

The EXITM directive causes immediate termination of a macro expansion. It is useful when used with the conditional assembly directive IF to terminate macro expansion when error conditions are detected.

The assembler does not allow a label with this directive.

See also: DUP, DUPA, DUPC, DUPF, MACRO

Example 5-42. EXITM Directive

```
CALC      MACRO  XVAL,YVAL
          IF  XVAL<0
          FAIL  'Macro parameter value out of range'
          EXITM                                          ; Exit macro
          ENDIF
          .
          .
          .
          ENDM
```

FAIL
Programmer Generated Error

Syntax

```
FAIL  [{str|exp}[,{str|exp},...]]
```

Description

The FAIL directive causes the assembler to output an error message. The total error count will be incremented as with any other error. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be optionally specified to describe the nature of the generated error.

The assembler does not allow a label with this directive.

See also: MSG, WARN

Example 5-43. FAIL Directive

```
          FAIL  'Parameter out of range'
```

FALIGN
Align Hardware Loop

Syntax

```
FALIGN
```

Description

The FALIGN directive aligns the address of the first instruction in a hardware loop, or the address of the target of a jump instruction, with the fetch set. The fetch set boundary is 16 bytes for the SC140 core.

The FALIGN directive may be applied on a per-loop basis. Alignment is performed when the size of the execution set overlaps the fetch set boundary. If the execution set starts at a nonaligned address, but fits into the current fetch set, alignment is not performed.

Alignment padding is done by issuing a single execution set containing the appropriate number of NOPs. Padding is done outside the loop, thereby eliminating cycle count overhead within the loop. A warning is issued if any NOPS are automatically added.

The FALIGN directive also has the effect of forcing the entire section to a 16-byte alignment. The alignment is preserved at link time even if the actual section starting location has moved.

See also: OPT (LPA option)

Example 5-44. FALIGN Directive

```
      ORG P:$100
      DOSETUP3 compute_alpha
      DOEN3  #5
      NOP
      NOP
      NOP
      LOOPSTART3
      FALIGN

compute_alpha
      ...
      LOOPEND3
```

GLOBAL
Global Section Symbol Declaration

Syntax

GLOBAL *symbol[,symbol,...]*

Description

The GLOBAL directive declares all the specified symbols within the current section as global. Symbols defined within a section are considered local, by default.

This directive is only valid when used within a program block bounded by the SECTION and ENDSEC directives.

The assembler does not allow a label with this directive.

See also: SECTION

Example 5-45. GLOBAL Directive

```
        SECTION IO
        GLOBAL LOOPA                    ; LOOPA will be globally accessible
        .                               ; by other files
        .
        .
        ENDSEC
```

GSET
Set Global Symbol to a Value

Syntax

```
label      GSET expression
           GSET label expression
```

Description

The GSET directive assigns the value of the expression in the operand field to the label. Labels defined using the GSET directive can have their values redefined in another part of the program, through the use of another GSET or SET directive.

The GSET directive is useful for resetting a global SET symbol within a section, where the SET symbol would otherwise be considered local. The expression in the operand field of a GSET must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

See also: DEFINE, EQU, SET

Example 5-46. GSET Directive

```
COUNT      GSET 0                                           ; Initialize count
```

HIMEM
Set High Memory Bounds

Syntax

```
HIMEM  P:expr[,...]
```

Description

The HIMEM directive establishes an absolute high memory bound for code and data generation. This directive may be used in absolute mode only.

The arguments for this directive are as follows:

`P:`	The P memory space.
`expr`	An absolute integer value within the address range of the machine.

The assembler does not allow a label with this directive.

See also: LOMEM

Example 5-47. HIMEM Directive

```
        HIMEM  P:$7FFF                            ; Set P run high mem bounds
```

IF
Conditional Assembly Directive

Syntax

```
IF expression
.
.
[ELSE]
.
.
ENDIF
```

Description

The IF-ENDIF directive pair defines the part of a program that is to be conditionally assembled. The `expression` must have an absolute integer result. It is considered true if it has a nonzero result, and false if it has a zero result. Because of the nature of the directive, `expression` must be known to the assembler on pass one (no forward references are allowed).

IF directives can be nested to any level. The ELSE directive always refers to the nearest previous IF directive, as does the ENDIF directive.

The following conditions determine whether the source statements bounded by the IF-ENDIF directives are included in the source file being assembled or are ignored:

If the ELSE directive is...	And the expression is...	Then the source statements between...
Not present	True (nonzero)	IF and ENDIF are included for assembly.
	False (zero)	IF and ENDIF are ignored.
Present	True (nonzero)	• IF and ELSE are included for assembly. • ELSE and ENDIF are ignored.
	False (zero)	• IF and ELSE are ignored. • ELSE and ENDIF are included for assembly.

The assembler does not allow a label with this directive.

See also: ENDIF

Example 5-48. IF Directive

```
IF BIG_ENDIAN
  MOVE.W  (r0)-,d1                           ; Start traceback from
ELSE                                         ;   state zero.
  MOVE.W  (r0),d1
ENDIF
```

INCLUDE
Include Secondary File

Syntax

```
INCLUDE  {string | <string>}
```

Description

The INCLUDE directive directs the assembler to read source statements from a secondary file identified by the `string` argument. The INLCUDE directive may be inserted into the source program at any point where the secondary file is to be included in the source input stream.

The file name specified in the `string` argument must be compatible with the operating system and can include a pathname. If no extension is given for the file name, a default extension of `.asm` is assumed.

The assembler's search path for the secondary file differs depending on whether the `string` or `<string>` syntax is used:

`string` If this syntax is used, the assembler searches for the secondary file in the following order:

1. The current directory if only a file name is given, or the specified directory if both a file name and a pathname are given.
2. All directories named with the `-i` command-line option.

`<string>` If this syntax is used, the assembler searches only the directories named with the `-i` command-line option. It does not search the current directory.

The assembler does not allow a label with this directive.

See also: MACLIB

Example 5-49. INCLUDE Directive

```
        INCLUDE  'headers/io.asm'                  ; Include the file io.asm
                                                   ; located in the headers
                                                   ; directory.
        INCLUDE  <data.asm>                        ; Do not look in the current
                                                   ; directory
```

LIST
List the Assembly

Syntax

```
        LIST
```

Description

The LIST directive prints the source listing, beginning at the point at which the LIST directive is encountered. The LIST directive is not printed, but the source lines following it is output to the source listing.

The source listing is printed, by default. If the `-oIL` option has been specified on the command line, the LIST directive is ignored when it is encountered in the source program.

The LIST directive increments a counter that is checked for a positive value and is symmetrical with respect to the NOLIST directive. Note the following sequence:

```
                            ; Counter value currently 1
        LIST                ; Counter value = 2
        LIST                ; Counter value = 3
        NOLIST              ; Counter value = 2
        NOLIST              ; Counter value = 1
```

The assembler will not disable the listing until it encounters another NOLIST directive.

The assembler does not allow a label with this directive.

See also: NOLIST, OPT

Example 5-50. LIST Directive

```
        IF  LISTON
        LIST                ; Turn the listing back on
        ENDIF
```

LOMEM
Set Low Memory Bounds

Syntax

```
LOMEM  P:expr[,...]
```

Description

The LOMEM directive establishes an absolute low memory bound for code and data generation. This directive may be used in absolute mode only.

The components of this directive are as follows:

`P`	The P memory space.
`expr`	An absolute integer value within the address range of the machine.

The assembler does not allow a label with this directive.

See also: HIMEM

Example 5-51. LOMEM Directive

```
          LOMEM P:$100                                   ; Set P run low mem bounds
```

MACLIB
Macro Library

Syntax

`MACLIB` *`pathname`*

Description

The MACLIB directive specifies the pathname of a directory that contains macro definitions. Each macro definition must be in a separate file, and the file name must be the same as the macro with a `.asm` extension. For example, `blockmv.asm` would be a file containing the definition of the macro, `blockmv`.

If the assembler encounters a directive in the operation field that is not contained in the directive or mnemonic tables, it searches the directory specified by `pathname` for a file of the unknown name (with the `.asm` extension added). If it finds such a file, the assembler saves the current source line, and opens the file for input as an INCLUDE file. When the end of the file is encountered, the source line is restored and processing resumes. Because the source line is restored, the processed file must have a macro definition of the unknown directive name, or else an error will result when the source line is restored and processed. However, the processed file is not limited to macro definitions, and can include any legal source code statements.

Multiple MACLIB directives may be used, in which case the assembler searches each directory in the order in which it is encountered.

The assembler does not allow a label with this directive.

See also: INCLUDE

Example 5-52. MACLIB Directive

```
        MACLIB  'macros/mymacs/'
```

MACRO
Macro Definition

Syntax

```
label      MACRO [dumarg[,dumarg,...]]
           .
           .
           source statements
           .
           .
           ENDM
```

Description

The required label is the symbol by which the macro will be called. If the macro has the same name as an existing assembler directive or mnemonic, a warning is issued.

The definition of a macro consists of three parts:

- Header, which assigns a name to the macro and defines the dummy arguments. This is the MACRO directive, its label, and list of dummy arguments.
- Body, which contains the pattern of standard source statements.
- Terminator, which is the ENDM directive.

The dummy arguments are symbolic names that the macro processor replaces with arguments when the macro is called. The following applies to dummy arguments:

- Each dummy argument must obey the same rules as symbol names. Dummy argument names that are preceded by a percent sign (%) are not allowed.
- Multiple dummy arguments must be separated by commas with no intervening spaces.

Macro definitions may be nested but the nested macro will not be defined until the primary macro is expanded.

See Chapter 6, "Macro Operations and Conditional Assembly," for more information on macros.

See also: DUP, DUPA, DUPC, DUPF, ENDM

Example 5-53. MACRO Directive

```
SWAP_SYM  MACRO REG1,REG2                          ; Swap REG1,REG2 using D0 as temp
          MOVE R\?REG1,D0
          MOVE R\?REG2,R\?REG1
          MOVE D0,R\?REG2
          ENDM
```

MSG
Programmer Generated Message

Syntax

```
    MSG   [{str|exp}[,{str|exp},...]]
```

Description

The MSG directive causes the assembler to output a message. The error and warning counts are not affected. The MSG directive is normally used in conjunction with conditional assembly directives for informational purposes. Assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be optionally specified to describe the nature of the message.

The assembler does not allow a label with this directive.

See also: FAIL, WARN

Example 5-54. MSG Directive

```
          MSG  'Generating sine tables'
```

NOLIST
Stop Assembly Listing

Syntax

```
        NOLIST
```

Description

The NOLIST directive inhibits the listing from being printed, beginning at the point at which the NOLIST directive is encountered. Neither the NOLIST directive or the source lines that follow it are printed.

The NOLIST directive decrements a counter that is checked for a positive value and is symmetrical with respect to the LIST directive. Note the following sequence:

```
                                   ; Counter value currently 1
        LIST                       ; Counter value = 2
        LIST                       ; Counter value = 3
        NOLIST                     ; Counter value = 2
        NOLIST                     ; Counter value = 1
```

The assembler will not disable the listing still until it encounters another NOLIST directive.

The assembler does not allow a label with this directive.

See also: LIST, OPT

Example 5-55. NOLIST Directive

```
        IF  LISTOFF
        NOLIST                                                    ; Turn the listing off
        ENDIF
```

OPT
Assembler Options

Syntax

```
OPT option[,option...]                        [; comment]
```

Description

The OPT directive designates one or more assembler options. Multiple options must be separated by commas with no intervening spaces.

The options that are used as arguments to this directive may also be designated on the command line with `-o`. For example, a memory utilization report may be requested by entering either of the following:

- `-omu` on the command line, or
- `OPT MU` in the source file.

Options can be grouped by function into the following types. Table 5-3 identifies the options within each type:

- Listing format control
- Reporting
- Message control
- Symbol
- Assembler operation

Table 5-3. Summary of OPT Options

Type	Option	Description
Listing Format Control	FC	Fold trailing comments
	FF	Enable form feeds for page ejects
	FM	Format messages
	PP	Pretty print the source listing
	RC	Enable relative comment spacing
Reporting	CEX	Print DC expansions
	CL	Print conditional assembly directives
	DXL	Expand DEFINE directive strings in listing
	HDR	Generate listing headers
	IL	Inhibit source listing
	MC	Print macro calls
	MD	Print macro definitions
	MEX	Print macro expansions
	MU	Print memory utilization report
	NL	Print conditional assembly and section nesting levels
	U	Print skipped conditional assembly lines

Table 5-3. Summary of OPT Options (Continued)

Type	Option	Description
Message Control	AE	Check address expressions
	MSW	Warn on memory space incompatibilities
	UR	Flag unresolved references
	W	Display warning messages
Symbol	DEX	Expand DEFINE symbols within quoted strings
	IC	Ignore case in symbol names
	SO	Write symbols to object file
Assembler Operation	AEC	Allow escape characters in DC directive
	BE	Enable big-endian mode
	CC	Enable cycle counts
	CK	Enable checksumming
	CM	Preserve comment lines within macros
	CONTCK	Continue checksumming
	DLD	Do not restrict directives in loops
	GL	Make all section symbols global
	INTR	Perform interrupt location checks
	LDB	Enable source listing debug
	LE	Enable little-endian mode
	LPA	Enable hardware loop alignment
	MI	Scan MACLIB directories for include files
	PS	Pack strings
	PSM	Enable programmable short addressing mode
	RSV	Check reserve data memory locations
	SVO	Preserve object file on errors

Option Descriptions

All options have a default condition, and some are reset to their default condition at the end of pass one. In the descriptions that follow, a parenthetical insert specifies *default* if the option represents the default condition, and *reset* if the option is reset to its default state at the end of pass one.

Many options have a corresponding negative version, in which the option is prefixed with ‘no’ and its meaning is reversed.

The OPT options are shown here in upper case; however, these options are not case sensitive.

`AE`	(default, reset) Check address expressions for appropriate arithmetic operations. For example, this will check that only valid add or subtract operations are performed on address terms.
`AEC`	Allow the standard set of C language escape characters within DC directives. This is not allowed with the DC directive, by default.
`BE`	Generate output for a big-endian target configuration. The default is a little-endian configuration.
`CC`	Enable cycle counts and clear total cycle count. Cycle counts are shown on the output listing for each instruction. Cycle counts assume a full instruction fetch pipeline and no wait states.

`CEX`	Print DC expansions.
`CK`	Enable checksumming of instruction and data values and clear cumulative checksum. The checksum value can be obtained using the @CHK() function. (See Section 3.7, “Functions.”)
`CL`	(default, reset) Print the conditional assembly directives.
`CM`	(default, reset) Preserve comment lines of macros when they are defined. Note that any comment line within a macro definition that starts with two consecutive semicolons (;;) is never preserved in the macro definition.
`CONTCK`	Re-enable checksumming of instructions and data. Does not clear cumulative checksum value.
`DEX`	Expand DEFINE symbols within quoted strings. Can also be done on a case-by-case basis using double-quoted strings.
`DLD`	Do not restrict directives in DO loops. The presence of some directives in DO loops does not make sense, including some OPT directive variations. This option suppresses errors on particular directives in loops.
`DXL`	(default, reset) Expand DEFINE directive strings in listing.
`FC`	Fold trailing comments. Any trailing comments that are included in a source line will be folded underneath the source line and aligned with the opcode field. Lines that start with the comment character will be aligned with the label field in the source listing. The FC option is useful for displaying the source listing on 80 column devices.
`FF`	Use form feeds for page ejects in the source listing.
`FM`	Format assembler messages so that the message text is aligned and broken at word boundaries.
`GL`	Make all section symbols global. This has the same effect as declaring every section explicitly GLOBAL. This option must be given before any sections are defined explicitly in the source file.
`HDR`	(default, reset) Generate listing header along with titles and subtitles.
`IC`	Ignore case in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined.
`IL`	Inhibit source listing. This option stops the assembler from producing a source listing.
`INTR`	(default, reset in absolute mode) Perform interrupt location checks. Certain DSP instructions may not appear in the interrupt vector locations in program memory. This option enables the assembler to check for these instructions when the program counter is within the interrupt vector bounds.
`LDB`	Use the source listing as the debug source file rather than the assembly language file. The `-l` command-line option to generate a source listing must be specified for this option to take effect.
`LE`	(default) Enable little-endian mode.
`LPA`	Enable hardware loop alignment for all subsequent hardware loops in the source file. The loop address must be aligned on a fetch set boundary in order to prevent an additional pipeline stall cycle from occurring each time the loop is executed. The LPA option allows the assembler to automatically pad the execution set prior to the loop start address with enough words (implemented as NOPs) to align the loop start address with a fetch set boundary. This prevents the additional pipeline stall cycle at each loop execution. The LPA option assumes that LOOPSTART notation is used; automatic alignment applies only to hardware loops with this notation.
`MC`	(default, reset) Print macro calls.

`MD`	(default, reset) Print macro definitions.
`MEX`	Print macro expansions.
`MI`	Scan the MACLIB directory paths for include files. The assembler ordinarily looks for included files only in the directory specified in the INCLUDE directory or in the paths given by the -i command line option. If the MI option is used, the assembler will also look for included files in any designated MACLIB directories.
`MSW`	(default, reset) Issue a warning on memory space incompatibilities.
`MU`	Include a memory utilization report in the source listing. This option must appear before any code or data generation.
`NL`	Display conditional assembly (IF-ELSE-ENDIF) and section nesting levels on listing.
`NOAE`	Do not check address expressions.
`NOAEC`	(default) Do not allow escape characters in DC directives.
`NOCC`	(default, reset) Disable cycle counts. Does not clear total cycle count.
`NOCEX`	(default, reset) Do not print DC expansions.
`NOCK`	(default, reset) Disable checksumming of instruction and data values.
`NOCL`	Do not print the conditional assembly directives.
`NOCM`	Do not preserve comment lines of macros when they are defined.
`NODEX`	(default, reset) Do not expand DEFINE symbols within quoted strings.
`NODLD`	(default, reset) Restrict use of certain directives in DO loop.
`NODXL`	Do not expand DEFINE directive strings in listing.
`NOFC`	(default, reset) Inhibit folded comments.
`NOFF`	(default, reset) Use multiple line feeds for page ejects in the source listing.
`NOFM`	(default, reset) Do not format assembler messages.
`NOHDR`	Do not generate listing header. This also turns off titles and subtitles.
`NOINTR`	(default, reset in relative mode) Do not perform interrupt location checks.
`NOLPA`	(default) Disable hardware loop alignment.
`NOMC`	Do not print macro calls.
`NOMD`	Do not print macro definitions.
`NOMEX`	(default, reset) Do not print macro expansions.
`NOMI`	(default, reset) Do not scan MACLIB directory paths for include files.
`NOMSW`	Do not issue warning on memory space incompatibilities.
`NONL`	(default, reset) Do not display nesting levels on listing.
`NOPP`	Do not pretty print the source listing. Source lines are sent to the source listing as they are encountered in the source, with the exception that tabs are expanded to spaces and continuation lines are concatenated into a single physical line for printing.
`NOPS`	Do not pack strings in DC directive. Individual bytes in strings will be stored one byte per word.
`NORC`	(default, reset) Do not space comments relatively.
`NOU`	(default, reset) Do not print the lines excluded from the assembly due to a conditional assembly directive.

`NOUR`	(default, reset) Do not flag unresolved external references.
`NOW`	Do not print warning messages.
`PP`	(default, reset) Pretty print (format) the source listing. The assembler attempts to align fields at a consistent column position without regard to source file formatting.
`PS`	(default, reset) Pack strings in DC directive. Individual bytes in strings will be packed into consecutive target words for the length of the string.
`RC`	Space comments relatively in listing fields. By default, the assembler places comments at a consistent column position in the source listing. This option allows the comment field to float; on a line containing only a label and opcode, the comment would begin in the operand field.
`SO`	Write symbol information to object file.
`SVO`	Preserve object file on errors. Normally any object file produced by the assembler is deleted if errors occur during assembly. This option must be given before any code or data is generated.
`U`	Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive.
`UR`	Generate a warning at assembly time for each unresolved external reference. This option works only in relocatable mode.
`W`	(default, reset) Print all warning messages.

The assembler does not allow a label with this directive.

Example 5-56. OPT Directive

```
         OPT CEX,MEX             ; Enable DC and macro expansions
         OPT CRE,MU              ; Print cross reference, memory utilization
```

ORG
Initialize Memory Space and Location Counters

Syntax

`ORG P``[`*`lc`*`]`**`:`**`[`*`exp`*`]`
`ORG P``[`**`(`***`ce`***`)`**`]`**`:`**`[`*`exp`*`]`

Description

The ORG directive is used in absolute mode. It has multiple functions, as follows:

- Sets absolute addresses
- Sets the memory space (this is always P for the SC140 DSP core)
- Designates which counter to use as the location counter, and assigns it an initial value

The arguments for this directive are as follows:

`P`	The P memory space.
`lc`	The location counter associated with the P memory space. Valid values are L (counter 1) or H (counter 2). If neither L or H are specified, the default counter (counter 0) is used. Counters are useful for providing mnemonic links among individual memory blocks. The names of the H, L and default counters are symbolic only; the assembler does not perform checks to insure that the value assigned to the H counter is greater than that of the L counter
`ce`	A non-negative absolute integer expression representing the counter number. Must be enclosed in parentheses and should not exceed the value 65535.
`exp`	The initial value to assign to the location counter. If `exp` is not specified, then the last value of the counter will be used.

The assembler does not allow a label with this directive.

Example 5-57. ORG Directive

```
        ORG  P:$1000             ; Sets the memory space to P. Selects the
                                 ; default counter (counter 0) associated
                                 ; with P space, and initializes it to $1000.

        ORG  PH:                 ; Sets the memory space to P. Selects the
                                 ; H location counter (counter 2) associated with P
                                 ; space. The H counter is not initialized, so its
                                 ; last value will be used.
```

PAGE
Top of Page/Size Page

Syntax

PAGE [*pagewidth*[,*pagelength*,*blanktop*,*blankbtm*,*blankleft*]]

Description

The PAGE directive serves two functions:

- Used with no arguments, it advances the source listing to the top of the next page. When used in this manner, the PAGE directive is not printed in the listing.
- Used with arguments, it sets the size and margins of the source listing. In this case, the PAGE directive is printed in the listing.

The arguments of the PAGE directive are described below. The arguments may be any positive absolute integer expression, and must be separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas.

`pagewidth`	Number of columns per line. The default is 80, the minimum is 1, and the maximum is 255.
`pagelength`	Total number of lines per page. The default is 66, the minimum is 10, and the maximum is 255. As a special case, a page length of 0 turns off all headers, titles, subtitles, and page breaks.
`blanktop`	Number of blank lines at the top of the page. The default is 0, the minimum is 0 and the maximum must maintain the relationship: `blanktop+blankbtm ≤ pagelength-10`.
`blankbtm`	Number of blank lines at the bottom of the page. The default is 0, the minimum is 0, and the maximum must maintain the relationship: `blanktop+blankbtm ≤ pagelength-10`.
`blankleft`	Number of blank columns at the left of the page. The default is 0, the minimum is 0, and the maximum must maintain the relationship: `blankleft < pagewidth`.

The assembler does not allow a label with this directive.

Example 5-58. PAGE Directive

```
        PAGE  132,,3,3          ; Set width to 132 columns, lines to default, and
                                ; top and bottom margins to 3 lines
        ...

        PAGE                    ; Eject the page
```

PMACRO
Purge Macro Definition

Syntax

```
        PMACRO  symbol[,symbol,...]
```

Description

The PMACRO directive purges the named macro definition from the macro table, reclaiming the macro table space.

The assembler does not allow a label with this directive.

See also: MACRO

Example 5-59. PMACRO Directive

```
        PMACRO  MAC1,MAC2
```

PRCTL
Send Control String to Printer

Syntax

`PRCTL` `{`*`exp`*`|`*`string`*`},...`

Description

The PRCTL directive concatenates its arguments and sends them to the source listing; the directive line itself is not printed unless there is an error. The arguments accepted by PRCTL consist of the following:

`exp`	A byte expression which may be used to encode non-printing control characters, such as ESC.
`string`	An assembler string which may be of arbitrary length, up to the maximum assembler-defined limits.

PRCTL may appear anywhere in the source file; the control string will be output at the corresponding place in the source listing. However, if a PRCTL directive is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. In this manner, a PRCTL directive can be used to restore a printer to a previous mode after printing is done. Similarly, if the PRCTL directive appears as the first line in the first input file, the control string will be output before page headings or titles.

The `-l` command-line option must be specified in order for the PRCTL directive to work; otherwise, it is ignored. See Section 2.1, "Invoking the Assembler," for more information on the -l option.

The assembler does not allow a label with this directive.

Example 5-60. PRCTL Directive

```
        PRCTL $1B,'E'                                ; Reset HP LaserJet printer
```

RADIX
Change Input Radix for Constants

Syntax

```
RADIX  expression
```

Description

The RADIX directive changes the input base of constants to the result of the specified expression. The absolute integer expression must evaluate to one of the legal constant bases: 2, 10, or 16. The default radix is 10.

The RADIX directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix prefix for base 10 numbers is the grave accent (`). Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the RADIX directive is encountered.

The assembler does not allow a label with this directive.

Example 5-61. RADIX Directive

```
_RAD10   DC 10                                       ; Evaluates to hex A
         RADIX 2
_RAD2    DC 10                                       ; Evaluates to hex 2
         RADIX `16
_RAD16   DC 10                                       ; Evaluates to hex 10
         RADIX 3                                     ; Bad radix expression
```

SECFLAGS
Set ELF Section Flags

Syntax

SECFLAGS `flag[,flag...]`

Description

The SECFLAGS directive sets the flag bits for the current section. Each `flag` describes an attribute of the section, and may be any of the following:

`write`	The section contains data that is writable when loaded.
`alloc`	The section occupies space in memory when loaded.
`execinstr`	The section contains executable instructions.
`nowrite`	The section contains data that is not writable when loaded.
`noalloc`	The section does not occupy space in memory when loaded.
`noexecinstr`	The section does not contain executable instructions.

Multiple `flag` arguments must be separated by commas with no intervening spaces.

Sections named .text, .data, .rodata, or .bss have predefined types and flags, as shown in Table 5-4 on page 5-69. A section with any other name is assumed to be a code section, and its type and flags default to those associated with a .text section. These default flags may be redefined as necessary using the SECFLAGS directive.

The assembler does not allow a label with this directive.

See also: SECTYPE, SECTION

Example 5-62. SECFLAGS Directive

```
;; Begin a data section. Is seen as .text section by assembler, with default
::  flags of nowrite, alloc, execinstr.

      SECTION .data_input2
      SECFLAGS write,alloc,noexecinstr    ;Set flags for a data section
      ...
      ENDSEC
```

SECTION
Start Section

Syntax

```
SECTION  symbol  [GLOBAL]
.
.
section source statements
.
.
ENDSEC
```

Description

The SECTION directive defines the beginning of a section. The arguments to this directive are as follows:

`symbol`	The name assigned to the section
`GLOBAL`	A qualifier that declares all symbols defined within the section to be global

Sections. Sections are the basic grouping for relocation of code and data blocks. Code or data inside a section is independently relocatable within the memory space to which it is bound. Sections can be:

- **Nested to any level.**

  When the assembler encounters a nested section, the current section is stacked and the new section is used. When the ENDSEC directive of the nested section is encountered, the assembler restores the old section and uses it. The ENDSEC directive always applies to the most previous SECTION directive.
- **Split into separate parts.**

  The section name can be used multiple times with SECTION and ENDSEC directive pairs. This allows the program source to be arranged in an arbitrary manner (for example, all statements that reserve P space storage locations grouped together).

For every section defined in the source, a location counter is allocated for the P memory space. This counter maintains offsets of data and instructions relative to the beginning of the section. At link time, sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined, so that each section can be relocated as a unit.

Section Names. A section may be given any name. However, the assembler recognizes the names of several default ELF sections, whose types and attributes are predefined (see Table 5-4). When these section names are used, the assembler defaults to the predefined type and flags associated with the name.

The assembler assumes that sections with other names are code (.text) sections, and sets the type and flags accordingly. If the section is not a code section, then the programmer must override the type and flag settings, as appropriate, using the SECTYPE and SECFLAGS directives, respectively.

Table 5-4. Predefined ELF Sections

Section	Contents	Type	Attributes
.text	Program code	PROGBITS	ALLOC, EXECINSTR
.data	Initialized data	PROGBITS	ALLOC, WRITE
.rodata	Read-only, initialized data	PROGBITS	ALLOC
.bss	Uninitialized data	NOBITS	ALLOC, WRITE

Names of certain specialized ELF sections are reserved and should not be used. These section names are listed in Table 5-5.

Table 5-5. Reserved Section Names

.debug_abbrev	.debug_pubname	.rel.line
.debug_info	.default	.rel.text
.debug_aranges	.hash	.shstrtab
.debug_line	.line	.strtab
.debug_macro	.note	.symtab

Symbol Binding. Symbols defined outside a section are considered global. They can satisfy an outstanding reference in the current file at assembly time, or in any file at link time. Global symbols may be referenced freely from inside or outside any section, as long as the global symbol name does not conflict with another symbol of the same name.

Symbols defined within a section are considered local, by default. Any reference to a local symbol can be satisfied in the file in which it is defined. Local symbols within sections may be declared global as follows:

- An individual symbol may be declared global by using the GLOBAL directive (see page 5-46).
- All symbols within a section (up to the next ENDSEC directive) may be declared global by using the GLOBAL qualifier with the SECTION directive (see Example 5-63).

The assembler does not allow a label with this directive.

See also: ORG, GLOBAL

Example 5-63. SECTION Directive

```
          SECTION TABLES GLOBAL                         ; TABLES is the section name.
                                                        ; All symbols in TABLES are
                                                        ; declared global.
```

SECTYPE
Set ELF Section Type

Syntax

SECTYPE {progbits|nobits}

Description

The SECTYPE directive defines a section's type. Valid types for this directive are as follows:

`progbits`	Indicates that the section contains program contents, including code and data.
`nobits`	Indicates that the section has no contents and does not occupy space in the file. The assembler discards anything contained in sections identified as `nobits`.

Sections named .text, .data, .rodata, or .bss have predefined types and flags, as shown in Table 5-4 on page 5-69. A section with any other name is assumed to be a code section, and its type and flags default to those associated with a .text section. If this default type is not appropriate, it may be redefined using the SECTYPE directive.

The assembler does not allow a label with this directive.

See also: SECFLAGS, SECTION

Example 5-64. SECTYPE Directive

```
;; Begin an uninitialized data section. Is viewed as .text section by
;;  assembler, with default type of progbits.

      SECTION .data_output
      SECFLAGS write,alloc,noexecinstr    ;Set flags for a data section
      SECTYPE nobits                      ;Set type for a .bss section
      ...
      ENDSEC
```

SET
Set Symbol to a Value

Syntax

```
label    SET  expression
         SET  label  expression
```

Description

The SET directive assigns the value of the expression to the label. The values of labels defined with the SET directive may be redefined in another part of the program through the use of another SET directive.

The SET directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a SET directive must be absolute and cannot include a symbol that is not yet defined (that is, no forward references are allowed).

See also: DEFINE, EQU, GSET

Example 5-65. SET Directive

```
COUNT    SET 0                                             ; Initialize count
```

SIZE
Set Symbol Size

Syntax

```
SIZE  symbol,expression
```

Description

The SIZE directive sets the size of the specified symbol to the value represented by `expression`.

The SIZE directive may occur anywhere in the source file unless the specified symbol is a function. In this case, the SIZE directive must occur after the function has been defined. This is illustrated in Example 5-66.

See also: TYPE

Example 5-66. SIZE Directive

```
_main:
  .
  .
  RTS
  SIZE  _main,(*-_main)
```

STITLE
Initialize Program Sub-Title

Syntax

```
        STITLE  [string]
```

Description

The STITLE directive initializes the program subtitle to the string specified in the operand field. The subtitle will be printed on the top of all succeeding pages of the source listing, following the title if one is present, until another STITLE directive is encountered. An STITLE directive with no string argument causes the current subtitle to be blank.

The subtitle is initially blank. The STITLE directive will not be printed in the source listing.

The assembler does not allow a label with this directive.

See also: TITLE

Example 5-67. STITLE Directive

```
        STITLE  'COLLECT SAMPLES'
```

TITLE
Initialize Program Title

Syntax

```
          TITLE  [string]
```

Description

The TITLE directive initializes the program title to the string specified in the operand field. The program title will be printed after the banner at the top of all succeeding pages of the source listing until another TITLE directive is encountered. A TITLE directive with no string argument causes the current title to be blank.

The title is initially blank. The TITLE directive will not be printed in the source listing.

The assembler does not allow a label with this directive.

See also: STITLE

Example 5-68. TITLE Directive

```
          TITLE  'FIR FILTER'
```

TYPE
Set Symbol Type

Syntax

Label **TYPE** *typeid*

Description

The TYPE directive sets a symbol’s type to the specified value in the ELF symbol table. Valid symbol types are as follows:

`FUNC`	The symbol is associated with a function or other executable code.
`OBJECT`	The symbol is associated with an object such as a variable, an array, or a structure.
`FILE`	The symbol name represents the file name of the compilation unit.

See also: SIZE

Example 5-69. TYPE Directive

```
Afunc   TYPE  FUNC                  ; The symbol Afunc is of type STT_FUNC
```

UNDEF
Undefine DEFINE Symbol

Syntax

```
        UNDEF   symbol
```

Description

The UNDEF directive releases the substitution string associated with `symbol`, and `symbol` will no longer represent a valid DEFINE substitution. See the DEFINE directive for more information.

The assembler does not allow a label with this directive.

See also: DEFINE

Example 5-70. UNDEF Directive

```
        UNDEF   DEBUG           ; Undefine the debug substitution string
```

WARN
Programmer-Generated Warning

Syntax

```
        WARN  [{str|expr}[,{str|expr},...]]
```

Description

The WARN directive causes the assembler to output a warning message, and increment the total warning count as with any other warning. The WARN directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. Assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be optionally specified to describe the nature of the generated warning.

The assembler does not allow a label with this directive.

See also: FAIL, MSG

Example 5-71. WARN Directive

```
        WARN  'parameter too large'
```

Chapter 6
Macro Operations and Conditional Assembly

Macros provide a shorthand method for handling a repeated pattern of code or group of instructions. Programmers can define the pattern as a macro, and then call the macro at the points in the program where the pattern would repeat.

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly for a given occurrence of the instruction group. Macros accommodate either case by allowing programmers to designate selected fields of any statement within a macro as variable. Thereafter, simply by invoking the macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

6.1 Macro Definition

The first step in using a macro is to define it, either in the source file or in a macro library (macro libraries are discussed in Section 6.4). A macro definition, illustrated in Figure 6-1, consists of three parts:

- Header, which assigns a name to the macro and optionally defines the dummy arguments. The header consists of the MACRO directive.
- Body, which contains the code or instructions to be substituted when the macro is invoked.
- Terminator, which signifies the end of the macro definition. This consists of the ENDM directive.

```
Header →      label    MACRO [dumarg[,dumarg...]          [; comment]
                       .
Body {                 source statements
                       .
Terminator →           ENDM
```

Figure 6-1. Macro Definition

The header, or MACRO directive, consists of the following:

`label`	The name by which the macro will be invoked. If the macro name is the same as an existing assembler directive or mnemonic opcode, the assembler replaces the directive or mnemonic opcode with the macro and issues a warning. This replacement does not occur for definitions from macro libraries.
`dumarg`	One or more optional symbolic names that, upon execution of the macro call, are replaced by the arguments defined by the macro call. Each dummy argument must obey the same rules as global symbol names. Dummy argument names that are preceded by a percent sign (%) are not allowed. Multiple dummy arguments must be separated by commas.
`comment`	An optional comment.

An important feature in defining a macro is the use of macro calls within the macro definition. The assembler processes such *nested* macro calls at expansion time only. The nested macro definition is not be processed until the primary macro is expanded. The macro must be defined before its appearance in a source statement operation field.

6.2 Macro Call

A macro is invoked with a macro call. The effect of the macro call is to produce in-line code to perform a predefined function. The code is inserted in the normal flow of the program so that the generated source statements are executed with the rest of the program each time the macro is called.

To invoke a macro, construct a source statement with the following format:

```
[label]    macro [arg[,arg...]]                                  [; comment]
```

where:

`label`	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
`macro`	The name of the macro. This must be in the operation field.
`arg`	One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
`comment`	An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the dummy arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.
- Macro arguments can be specified as quoted strings, although the assembler does not require the use of single quotes around macro argument strings. However, if an argument has an embedded comma or space, that argument must be surrounded by single quotes (`'`).

- A macro call argument can be declared null, but must be done so explicitly by:
  - — Entering delimiting commas in succession with no intervening spaces
  - — Terminating the argument list with a comma and omitting the rest of the argument list
  - — Declaring the argument as a null string
- No character is substituted in the generated statements that reference a null argument.

6.3 Macro Expansions

Macro expansions, or the source statements generated by the macro call, may contain substitutable arguments, and are relatively unrestricted as to type. They can be any processor instruction, most any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions that are applied to statements written by the programmer.

6.4 Macro Libraries

Macro libraries contain macro definitions. Each definition must be in a separate file, and the file name must be the same as the macro name with a `.asm` extension. For example, `blockmv.asm` would be the name of the file that contained the definition of the macro, `blockmv`.

The MACLIB directive specifies the pathname of a directory that contains macro definitions. If a MACLIB directive has been specified in the source code and the assembler encounters a name in the operation field that is not a previously defined macro or is not contained in the directive or mnemonic tables, the directory specified in the MACLIB directive will be searched for a file of that name (with the `.asm` extension added). If such a file is found, the current source line will be saved, and the file will be opened for input as an INCLUDE file. When the end of the file is encountered, the source line is restored and processing is resumed.

Because the source line is restored, the processed file must have a macro definition of the unknown name, or an error will result when the source line is restored and processed. However, the processed file is not limited to macro definitions, and can include any legal source code statements. Multiple MACLIB directives may be given, in which case the assembler will search each directory in the order in which they were specified.

6.5 Dummy Argument Operators

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. These operators, listed in Table 6-1, can be used for text concatenation, numeric conversion, and string handling. For more information, see Section 5.1, "Assembler Significant Characters."

Table 6-1. Macro Dummy Argument Operators

Operator	Name	Description
\	Macro Argument Concatenation Character	Concatenates a macro dummy argument with adjacent alphanumeric characters
?	Return Value of Symbol	Substitutes the ?*symbol* sequence with a character string representing the decimal value of the symbol
%	Return Hex Value	Converts the %*symbol* sequence to a character string representing the hexadecimal value of the symbol
"	Macro String Delimiter	Allows the use of macro arguments as literal strings.
^	Macro Local Label Override	Causes local labels in its term to be evaluated at normal scope rather than at macro scope

6.6 Macro Directives

The DUP, DUPA, DUPC, and DUPF directives are specialized macro forms, which can be thought of as a simultaneous definition and call of an unnamed macro. The source statements defined with any of these directives follow the same rules as macro definitions, including (in the case of DUPA, DUPC, and DUPF) the dummy operator characters described previously.

For a detailed description of these directives, refer to Section 5.2, "Assembler Directives."

6.7 Conditional Assembly

Conditional assembly facilitates the writing of comprehensive source programs that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols through the DEFINE, SET, and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the given conditions. The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment. (See Section Section 3.7, "Functions," for more information on the assembler built-in functions.)

Conditional directives can also be used within a macro definition to ensure that arguments fall within a range of allowable values at expansion time. In this way, macros become self-checking and can generate error messages to any desired level of detail.

A section of a program that is to be conditionally assembled must be bounded by an IF-ENDIF directive pair, as follows:

```
IF expression
.
source statements
.
[ELSE]
.
source statements
.
ENDIF
```

The following conditions determine whether the source statements bounded by the IF-ENDIF directives are included in the source file being assembled or are ignored:

If the ELSE directive is...	And expression is...	Then the source statements between...
Not present	True (non-zero)	IF and ENDIF are included for assembly.
	False (zero)	IF and ENDIF are ignored.
Present	True (non-zero)	• IF and ELSE are included for assembly. • ELSE and ENDIF are ignored.
	False (zero)	• IF and ELSE are ignored. • ELSE and ENDIF are included for assembly.

Chapter 7
SC100 Linker

The SC100 Linker combines object files and archive files, relocates their data, adjusts the symbol references, and generates an absolute executable file.

The linker allows the programmer to break up a large program into more manageable modules, which may be assembled or compiled separately. These modules can then be link edited to produce an absolute module of the complete program. If a problem arises, only the module with the problem need be edited and reassembled. The programmer can then relink the updated relocatable object module and the other previously created object modules to produce a new executable file.

7.1 Invoking the Linker

The command line to invoke the linker is as follows:

`sc100-ld` `[`*`option ...`*`]` *`file`* `...`

where:

`option`	One or more of the options listed in Table 7-1. Options and their arguments are case sensitive. Spacing must be maintained as shown; for example, a space is required between the `-c` option and its argument, but is not allowed with the `-l` option.
`file`	One or more relocatable object files. The files are processed in the order listed.

Table 7-1. SC100 Linker Options

Option	Description
`-c` *`commandfile`*	Specifies the name and location of the linker command file
`-l`*`archive`*	Links against the specified archive, located in the standard search paths
`-L`*`searchdir`*	Adds the specified directory to the standard search paths
`-M`	Outputs a map file to the standard output
`-Map` *`mapfile`*	Outputs a map file to the specified file
`-o` *`outfile`*	Assigns the specified file name, exactly as entered, to the executable object file. The default file name is `a.eld`
`-S`	Strips debug information from the executable object file
`-s`	Strips all symbol information from the executable object file
`-V`	Prints the current version of the linker
`-W`	Suppresses warnings
`-x`	Removes all local symbols from the symbol table

7.2 Linker Processing

The input to the linker is a set of relocatable object files produced by the SC100 Assembler. The term *relocatable* means that the data in the module has not yet been assigned to absolute addresses in memory. Instead, each different section is assembled as though it started at relative address 0.

During relocation, the linker assigns each segment to an absolute memory address, places the code and data from each segment into the proper location in the executable file, adjusts the segment addresses, and resolves symbol references.

The linker makes three partial passes over the input data. Figure 7-1 highlights the linker operations in each pass.

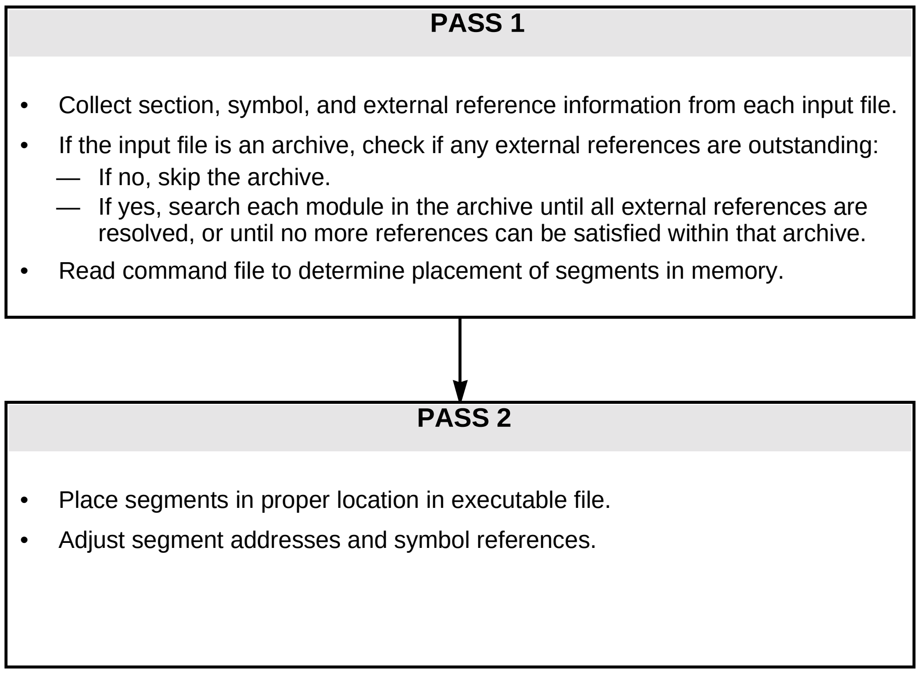

PASS 1
• Collect section, symbol, and external reference information from each input file.
• If the input file is an archive, check if any external references are outstanding:
— If no, skip the archive.
— If yes, search each module in the archive until all external references are resolved, or until no more references can be satisfied within that archive.
• Read command file to determine placement of segments in memory.
PASS 2
• Place segments in proper location in executable file.
• Adjust segment addresses and symbol references.

Figure 7-1. Linker Operation

7.3 Linker Command File

The linker command file contains a list of linker directives that allow a programmer to define a memory layout, allocate segments, and specify the program’s entry point.

A command file is shown in Example 7-1. The directives shown in this example are explained in Section 7.4, “Linker Directives.”

Example 7-1. Linker Command File

```
.provide DataSize, 0x8000
.provide CodeStart, 0x8000
.provide StackStart, 0x20000
.provide TopOfMemory, 0x7FFF0
.provide SR_Setting, 0xE40008

.memory 0, 0xFFFFF
.reserve StackStart, TopOfMemory
.reserve 0xFFFF0000, 0xFFFFFFFF
.entry IntVec

.org 0
.segment .intvec, "crtsc4*"

.org 0x100
.segment .data, ".data", ".bss"

.org CodeStart
.segment .text, ".text"
```

7.4 Linker Directives

The linker supports the directives listed in Table 7-2. These directives are described in the following sections.

Table 7-2. Summary of Linker Directives

Directive	Description
.entry	Defines the program’s entry point
.memory	Specifies a region in memory available for linking
.org	Specifies a starting address to begin linking segments
.provide	Inserts a symbol into the symbol table of the executable file
.reserve	Specifies a region in memory that is not available for linking
.segment	Specifies which sections to link at the current location counter

.entry
Set Program Entry Point

Syntax

```
.entry expression
```

Description

The .entry directive assigns the address at which to begin executing the program. The beginning address is specified by `expression`.

Example 7-2. .entry Directive

```
.entry 0x8000
```

.memory
Define Available Region

Syntax

```
.memory lo_addr, hi_addr
```

Description

The .memory directive defines a region in memory that is available for linking. The arguments, `lo_addr` and `hi_addr`, are expressions that set the region's low address and high address, respectively.

Example 7-3. .memory Directive

```
.memory 0, 0xFFFFF
```

.org
Set Linking Address

Syntax

```
.org expression
```

Description

The .org directive specifies a starting address for linking segments.

This directive is used in combination with the .segment directive. All .segment directives occurring after this directive but before the next .org are placed consecutively in memory, beginning at the address specified by `expression`.

In Example 7-4, the .org directive instructs the linker to begin linking the .text segment at the value represented by the `CodeStart` variable.

Example 7-4. .org Directive

```
.org CodeStart
.segment .text, ".text"
```

.provide
Insert ELF Symbol

Syntax

```
  .provide symbol, expression
```

Description

The .provide directive creates a global, absolute symbol in the symbol table of the executable file. The arguments to this directive are as follows:

`symbol`	Name of the symbol to be inserted into the symbol table.
`expression`	Value of the specified symbol.

Example 7-5. .provide Directive

```
.provide StackStart, 0x20000
.provide ROMStart, 0x7FFF0
.provide SR_Setting, 0xE40008
```

.reserve
Define Unavailable Region

Syntax

```
.reserve lo_addr, hi_addr
```

Description

The .reserve directive defines a region in memory that is *not* available for linking. The arguments, `lo_addr` and `hi_addr`, are expressions that set the region's low address and high address, respectively.

Example 7-6. .reserve Directive

```
.reserve StackStart, TopOfMemory
```

.segment
Define Segment

Syntax

.segment *seg_name*, *"section_pattern"*[,*"section_pattern"...*]

Description

The .segment directive specifies which sections to link at the current location counter. The linker combines all sections matching the listed pattern(s) into the named segment. Sections are added in the order specified by the pattern arguments.

Patterns may include asterisks for specifying an arbitrary character sequence. Each pattern must be enclosed in double quotes ("). Multiple patterns must be separated by commas.

The .segment directive is used in combination with the .org directive, which specifies where to begin placing segments in memory. For example, the .segment and .org directives shown in Example 7-4 instruct the linker to do the following:

- Construct a segment named TEXT and place in it all sections named .text, followed by all sections named .rodata.
- Construct a segment named DATA. Place the following sections into the DATA segment, in the order listed: all sections named .data, all sections whose names begin with 'mydata', and all sections named .bss.
- Place both segments consecutively in memory, beginning at address 0.

Example 7-7. .segment Directive

```
.org 0
.segment TEXT, ".text", ".rodata"
.segment DATA, ".data", "mydata*", ".bss"
```

The placement of these sections are shown in Figure 7-2. For purposes of this example, assume that `file1.eln` was linked before `file2.eln`.

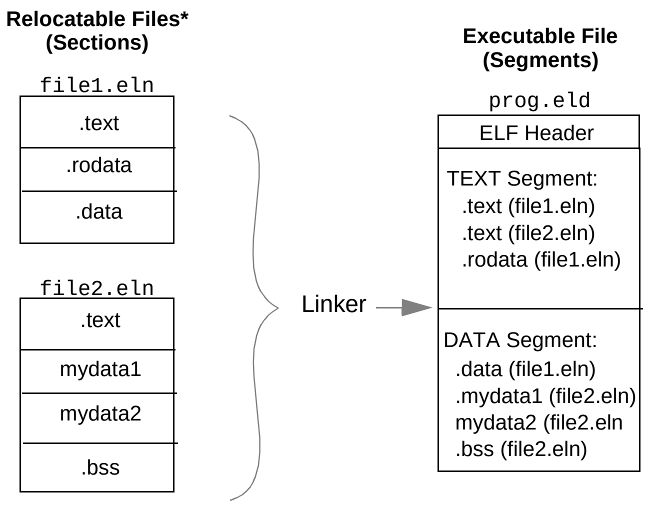


Figure 7-2. Placement of Sections Using .segment Directive

7.5 Linker Map File

A linker map file may be requested with the`-M` or `-Map` options. The map file lists all sections encountered in the linker input stream, and provides the following information for each:

- Total size of the section
- Address of the section after relocation
- Details of all section fragments that comprise the section. This includes for each fragment: its size, its address after relocation, the input file or library module in which it was found, and a listing of any symbols that it referenced.

Sections, section fragments, and symbols can be identified in the map file based on their placement in the Symbol field:

- A section always begins in column 1. The section name is preceded by the label “Section:”.
- A section fragment is placed on the first indent of the Symbol field. The fragment name is preceded by the label “Section:”, and is followed by the name, in parenthesis, of the input file or library module in which it was found.

  Each fragment has the same name as the section to which it belongs.
- A symbol is placed on the second indent of the Symbol field, and applies to the most previous fragment listed.

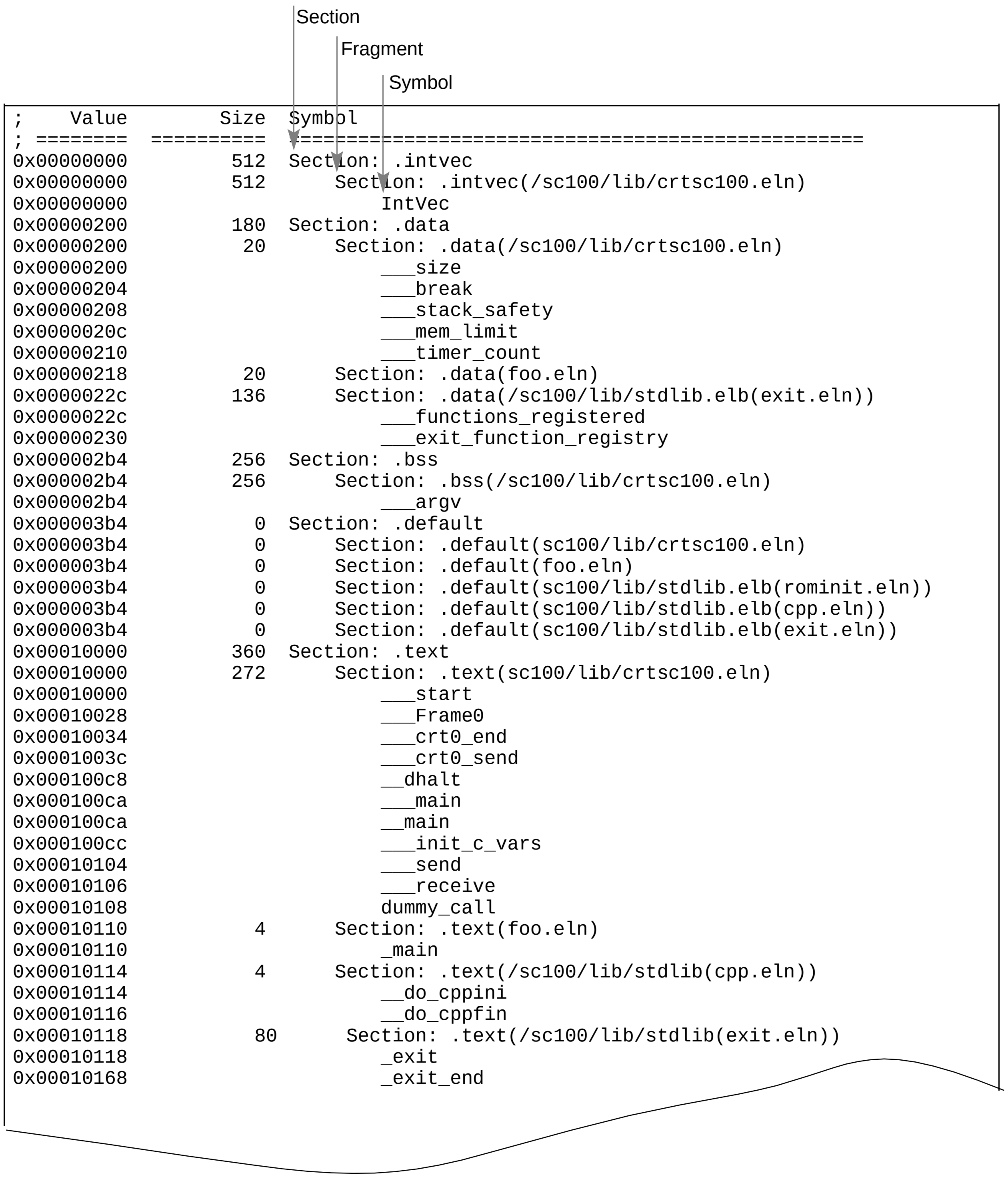
The partial map file shown in Figure 7-3 lists five sections: .intvec, .data, .bss, .default, and .text. Notice that the .text section has an address of $10000 after relocation, and is 360 bytes. This size represents the total of all the .text section fragments assimilated from separate input files. The .text section consists of four fragments, which break down as follows:

Table 7-3. Composition of .text Section

Fragment	Size	Originating Input File	Symbols Referenced Within Fragment

1	272	`/sc100/lib/crtsc100.eln`	`___start`
			`___Frame0`
			`___crt0_end`
			...

2	4	`foo.eln`	`_main`
3	4	`cpp.eln` module of the library, `/sc100/lib/stdlib.elb`	`__do_cppini` `__do_cppfin`
4	80	`exit.eln` module of the library, `/sc100/lib/stdlib.elb`	`_exit` `_exit_end`



```
;    Value        Size  Symbol
; ========  ==========  ==================================================
0x00000000         512  Section: .intvec
0x00000000         512      Section: .intvec(/sc100/lib/crtsc100.eln)
0x00000000                  IntVec
0x00000200         180  Section: .data
0x00000200          20      Section: .data(/sc100/lib/crtsc100.eln)
0x00000200                  ___size
0x00000204                  ___break
0x00000208                  ___stack_safety
0x0000020c                  ___mem_limit
0x00000210                  ___timer_count
0x00000218          20      Section: .data(foo.eln)
0x0000022c         136      Section: .data(/sc100/lib/stdlib.elb(exit.eln))
0x0000022c                  ___functions_registered
0x00000230                  ___exit_function_registry
0x000002b4         256  Section: .bss
0x000002b4         256      Section: .bss(/sc100/lib/crtsc100.eln)
0x000002b4                  ___argv
0x000003b4           0  Section: .default
0x000003b4           0      Section: .default(sc100/lib/crtsc100.eln)
0x000003b4           0      Section: .default(foo.eln)
0x000003b4           0      Section: .default(sc100/lib/stdlib.elb(rominit.eln))
0x000003b4           0      Section: .default(sc100/lib/stdlib.elb(cpp.eln))
0x000003b4           0      Section: .default(sc100/lib/stdlib.elb(exit.eln))
0x00010000         360  Section: .text
0x00010000         272      Section: .text(sc100/lib/crtsc100.eln)
0x00010000                  ___start
0x00010028                  ___Frame0
0x00010034                  ___crt0_end
0x0001003c                  ___crt0_send
0x000100c8                  __dhalt
0x000100ca                  ___main
0x000100ca                  __main
0x000100cc                  ___init_c_vars
0x00010104                  ___send
0x00010106                  ___receive
0x00010108                  dummy_call
0x00010110           4      Section: .text(foo.eln)
0x00010110                  _main
0x00010114           4      Section: .text(/sc100/lib/stdlib(cpp.eln))
0x00010114                  __do_cppini
0x00010116                  __do_cppfin
0x00010118            80     Section: .text(/sc100/lib/stdlib(exit.eln))
0x00010118                  _exit
0x00010168                  _exit_end
```

Figure 7-3. Linker Map File

Chapter 8
SC100 Utilities

The SC100 tools set includes several utilities that process or interpret ELF (executable and linking format) object files. This chapter describes the SC100 utilities, which include the following:

- Archiver
- Disassembler
- ELF File Dump Utility
- Name Utility
- Size Utility

8.1 Archiver

The archiver groups separate object files into a single file for linking or archival storage. Once an archive is created, new files may be added and existing files may be extracted, deleted, or replaced.

8.1.1 Invoking the Archiver

The archiver can be invoked using one of the following command lines. Options are case sensitive and are described in Table 8-1.

```
arsc100 -d [-v] archive file ...
arsc100 -p [-v] archive [file ...]
arsc100 -r [-c] [-u] [-v] archive file ...
arsc100 -t [-v] archive [file ...]
arsc100 -x [-v] archive [file ...]
```

where:

`archive`	Name of the archive to create or access.
`file`	One or more archive files. Multiple file names must be separated by blanks. The files are processed in the order listed.

Table 8-1. Archiver Options

Option	Description
-c	Suppresses the diagnostic message that is written to standard error by default when the archive is created. This option is used only with the `-r` command line.
-d	Deletes the listed `file`(s) from `archive`.
-p	Writes the contents of `file`(s) from `archive` to the standard output. If no files are listed, the contents of all files in the archive are written, in the order that they occur.

Table 8-1. Archiver Options (Continued)

Option	Description
-r	Replaces or adds `file`(s) to `archive`. If the archive does not exist, a new archive is created and, by default, a diagnostic message is written to standard error. New files are appended to the archive. Files that replace existing files do not change the order of the archive.
-t	Writes the archive's table of contents to the standard output, including all files specified on the command line. If no files are specified, all files in the archive are included in the list in the order that they appear in the archive.
-u	Updates older files. Used only with the `-r` command line, this option replaces files within the archive only if the corresponding `file` has a modification time that is at least as new as the modification time of the file within the archive.
-v	Gives verbose output. When used with the: • -d, -r, or -x command line, the verbose option writes a detailed file-by-file description of the archive creation and maintenance activity; • -p command line, the verbose option writes the name of the file to the standard output before writing the contents of the file to the standard output; • -t command line, the verbose option includes a long listing of information about the files within the archive.
-V	Prints the current version of the Archiver.
-x	Extracts `file`(s) from `archive`. If no files are listed, all files in the archive are extracted. The contents of the archive are not changed.

8.2 Disassembler

The sc100-dis utility disassembles the executable instructions in each ELF object file specified on the command line. Disassembly is written to the standard output.

8.2.1 Invoking the Disassembler

The command line to invoke the disassembler is as follows. Options are case sensitive:

```
sc100-dis [-x] {-i | file ...}
```

where:

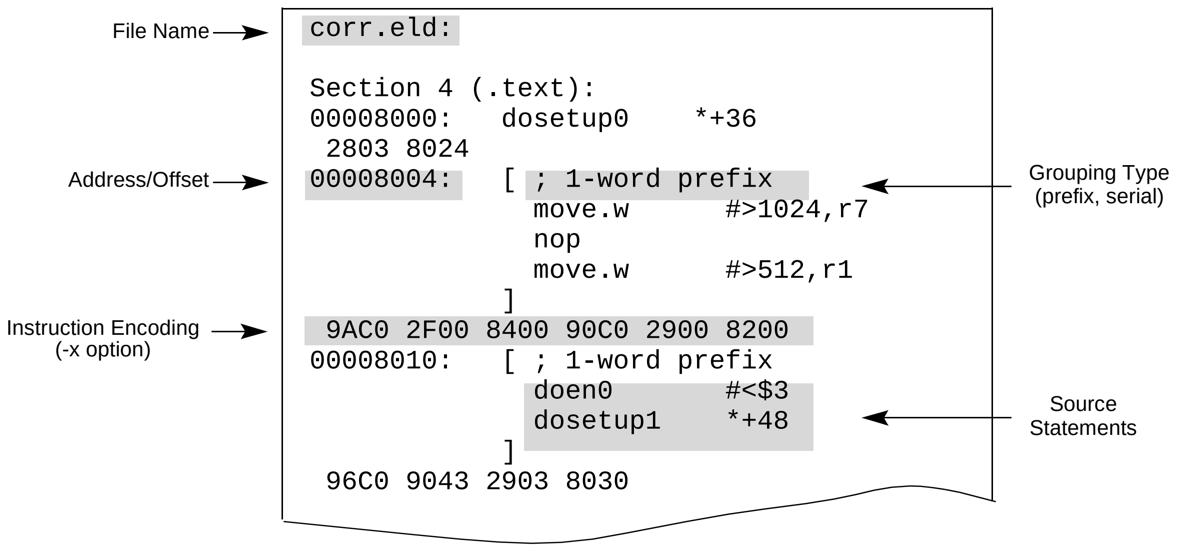
`file`	One or more ELF object files.

Table 8-2. Disassembler Utility Options

Option	Description
-i	Interactive mode. Reads hexadecimal instruction words entered from the standard input, and disassembles to the standard output, as usual.
-V	Prints the current version of the Disassembler.
-x	After disassembling an instruction or execution set, outputs its instruction words (in hexadecimal) to the standard output.

8.2.2 Disassembler Example

Figure 8-1 shows an example of disassembly output using the `-x` option, which adds instruction encoding information to the default output.



```
corr.eld:

Section 4 (.text):
00008000:   dosetup0    *+36
 2803 8024
00008004:   [ ; 1-word prefix
              move.w      #>1024,r7
              nop
              move.w      #>512,r1
            ]
 9AC0 2F00 8400 90C0 2900 8200
00008010:   [ ; 1-word prefix
              doen0       #<$3
              dosetup1    *+48
            ]
 96C0 9043 2903 8030
```

File Name
Address/Offset
Instruction Encoding (-x option)
Grouping Type (prefix, serial)
Source Statements

Figure 8-1. Disassembly Output with Instruction Encoding

8.3 ELF File Dump Utility

The ELF file dump utility outputs the formatted contents of header structures for each ELF object file specified on the command line. The default output is as follows:

- For an executable object file, the ELF header, all program headers, and all section headers
- For a relocatable object file, the ELF header and all section headers

Additional information may be added to the default output using command-line options, as described in Table 8-3.

8.3.1 Invoking the ELF File Dump Utility

The command line to invoke the ELF file dump utility is as follows:

`sc100-elfdump` `[`*`option ...`*`]` *`file ...`*

where:

`option`	One or more of the options listed in Table 8-3. Options are case sensitive.
`file`	One or more ELF object files.

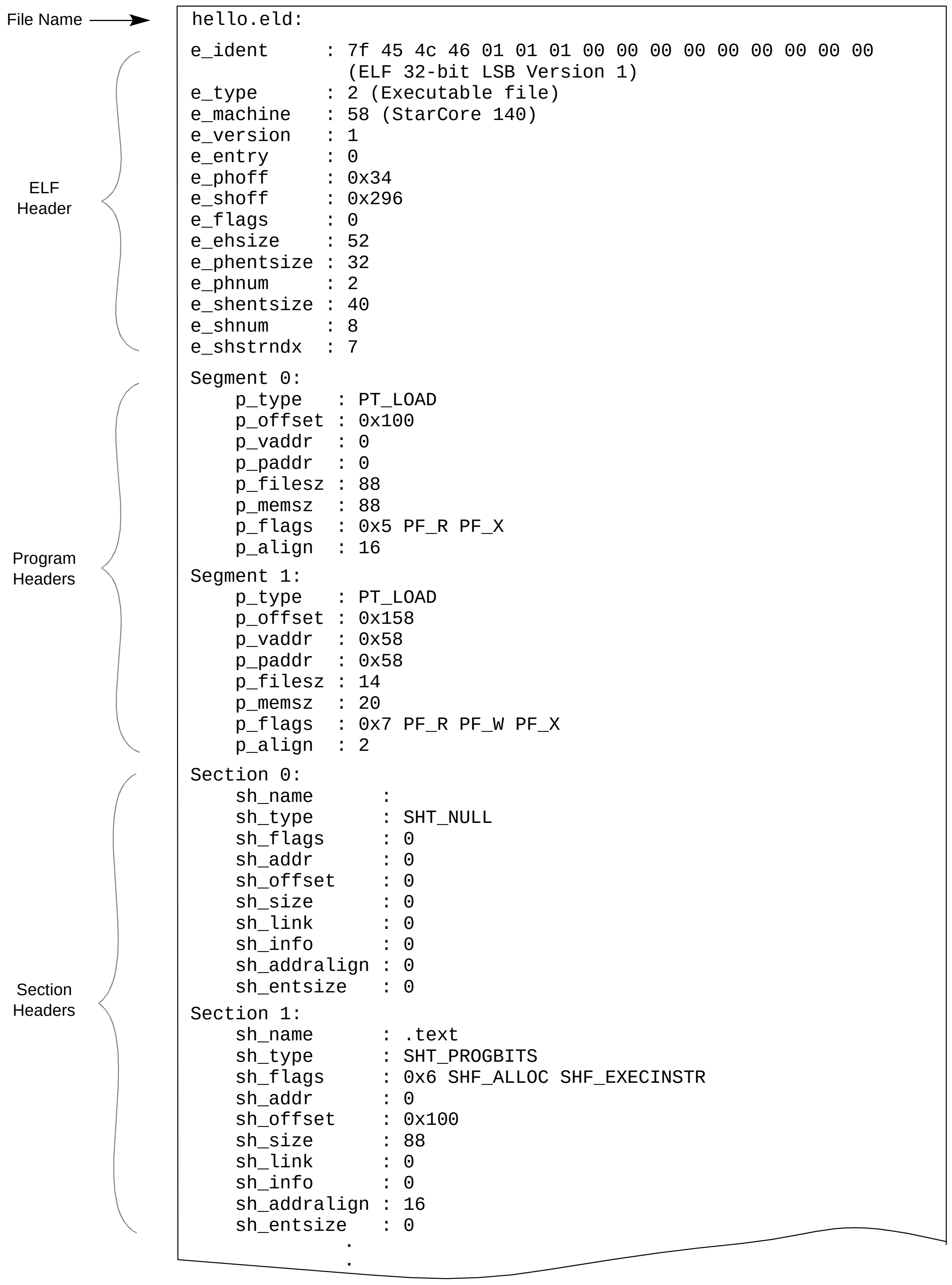
Table 8-3. ELF File Dump Utility Options

Option	Description
`-A`	Writes the contents of all segments
`-D`	Writes the contents of all PT_DYNAMIC segments[1]
`-I`	Writes the contents of all PT_INTERP segments[1]
`-L`	Writes the contents of all PT_LOAD segments
`-N`	Writes the contents of all PT_NOTE segments[1]
`-P`	Writes the contents of all PT_PHDR segments
`-S`	Writes the contents of all PT_SHLIB segments[1]
`-U`	Writes the contents of all segments of unknown type as hex dumps
`-X`	Dumps the section raw data from the specified file
`-a`	Writes the contents of all sections
`-b`	Writes the contents of all SHT_PROGBITS sections
`-d`	Writes the contents of all SHT_DYNAMIC sections[1]
`-h`	Writes the contents of all SHT_HASH sections[1]
`-n`	Writes the contents of all SHT_NOTE sections
`-r`	Writes the contents of all SHT_REL and SHT_RELA sections
`-s`	Writes the contents of all SHT_SHLIB sections[1]
`-t`	Writes the contents of all SHT_STRTAB sections
`-u`	Writes the contents of all sections of unknown type as hex dumps
`-V`	Prints the current version of the ELF File Dump utility
`-x`	Dumps all section contents in hex (as -u does for unknown types)
`-y`	Writes the contents of all SHT_SYMTAB sections
`-z`	Writes the contents of all SHT_DYNSYM sections[1]

1. Not applicable to the SC140 DSP core.

8.3.2 ELF File Dump Example

Figure 8-2 shows an example of the default output of the ELF file dump utility.



```
hello.eld:

e_ident     : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
              (ELF 32-bit LSB Version 1)
e_type      : 2 (Executable file)
e_machine   : 58 (StarCore 140)
e_version   : 1
e_entry     : 0
e_phoff     : 0x34
e_shoff     : 0x296
e_flags     : 0
e_ehsize    : 52
e_phentsize : 32
e_phnum     : 2
e_shentsize : 40
e_shnum     : 8
e_shstrndx  : 7

Segment 0:
    p_type   : PT_LOAD
    p_offset : 0x100
    p_vaddr  : 0
    p_paddr  : 0
    p_filesz : 88
    p_memsz  : 88
    p_flags  : 0x5 PF_R PF_X
    p_align  : 16
Segment 1:
    p_type   : PT_LOAD
    p_offset : 0x158
    p_vaddr  : 0x58
    p_paddr  : 0x58
    p_filesz : 14
    p_memsz  : 20
    p_flags  : 0x7 PF_R PF_W PF_X
    p_align  : 2

Section 0:
    sh_name      :
    sh_type      : SHT_NULL
    sh_flags     : 0
    sh_addr      : 0
    sh_offset    : 0
    sh_size      : 0
    sh_link      : 0
    sh_info      : 0
    sh_addralign : 0
    sh_entsize   : 0
Section 1:
    sh_name      : .text
    sh_type      : SHT_PROGBITS
    sh_flags     : 0x6 SHF_ALLOC SHF_EXECINSTR
    sh_addr      : 0
    sh_offset    : 0x100
    sh_size      : 88
    sh_link      : 0
    sh_info      : 0
    sh_addralign : 16
    sh_entsize   : 0
              .
              .
```

Figure 8-2. ELF File Dump Output

8.4 Name Utility

The name utility displays the symbolic information for each object file or object-file library specified on the command line.

If no symbolic information is available for a valid input file, the name utility reports that fact and exits without returning an error.

8.4.1 Invoking the Name Utility

The command line to invoke the name utility is as follows:

`sc100-nm` `[`*`option ...`*`]` *`file`* `...`

where:

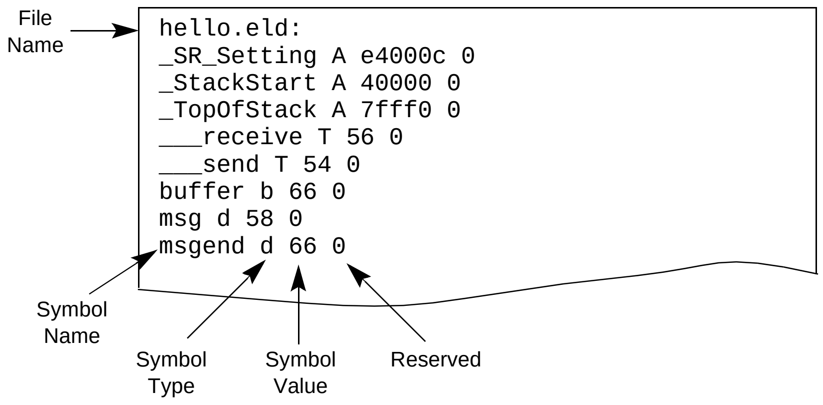
`option`	One or more of the options listed in Table 8-4. Options are case sensitive.
`file`	One or more ELF object files or object-file libraries.

Table 8-4. Name Utility Options

Option	Description
`-A`	Writes the full pathname or library name of an object on each line.
`-g`	Writes only external (global) symbol information. This option cannot be combined with the `-u` option.
`-P`	Writes the information in the POSIX.2 portable output format.
`-t` `{`**`d`** `|` **`o`** `|` **`x`**`}`	Writes each numeric value in the specified format, where: d—is the offset written in decimal o—is the offset written in octal x—is the offset written in hexadecimal (the default)
`-u`	Writes only undefined symbols. This option cannot be combined with the `-g` option.
`-v`	Sorts the output by value instead of by name.
`-V`	Prints the current version of the Name utility.

8.4.2 Name Utility Example

Figure 8-3 shows an example of the default output of the name utility.



```
hello.eld:
_SR_Setting A e4000c 0
_StackStart A 40000 0
_TopOfStack A 7fff0 0
___receive T 56 0
___send T 54 0
buffer b 66 0
msg d 58 0
msgend d 66 0
```

Figure 8-3. Name Utility Output

The symbol type may be one of the following. Where a letter is shown as either uppercase or lowercase, uppercase indicates a global symbol, and lowercase indicates a local symbol:

U	Undefined reference
A or a	Absolute symbol
B or b	BSS symbol
T or t	Text (code) symbol
D or d	Data symbol
R or r	Read-only data symbol
N	Debug symbol
?	Unknown symbol type or binding

8.5 Size Utility

The size utility outputs the sizes of sections, in bytes, for each object file specified on the command line.

By default, the size utility lists the totals of all text, rodata, data, and bss sections for each object file specified on the command line. Size information for individual sections and loadable segments can be obtained using command-line options, as described in Table 8-5.

8.5.1 Invoking the Size Utility

The command line to invoke the size utility is as follows:

`sc100-size` `[`*`option ...`*`]` *`file`* `...`

where:

`option`	One or more of the options listed in Table 8-5. Options are case sensitive.
`file`	One or more ELF object files.

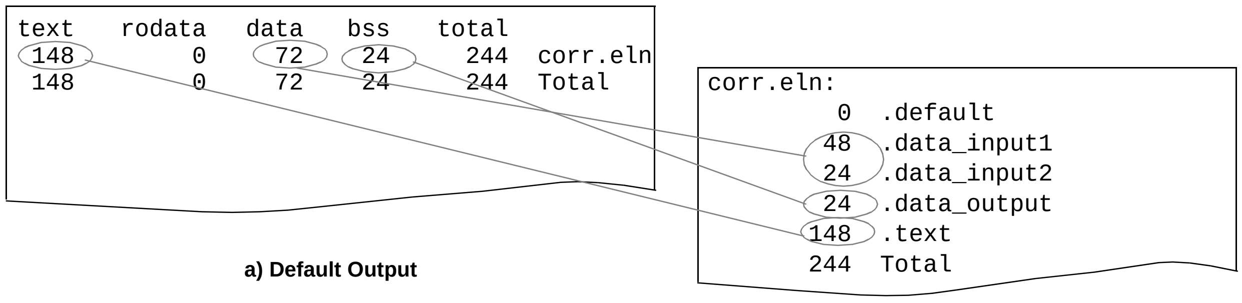
Table 8-5. Size Utility Options

Option	Description
`-l`	Outputs detailed information, listing the names and sizes of individual sections
`-n`	Outputs the size of executable sections which do not get loaded
`-p`	Outputs the size of all loadable segments (program view)
`-V`	Prints the current version of the Size utility

8.5.2 Size Utility Example

Figure 8-4 shows two examples of size utility output. Case a) shows the default output, which lists the totals of all text, rodata, data, and bss sections in the object file. Case b) shows the long listing format, which is generated using the `-l` option. This format lists the details of the individual sections that make up the totals shown in the default output.

For example, in Case a), notice that the total of all data sections in the `corr.eln` object file is 72 bytes. The sections that contribute to this total are listed in Case b). They are .data_input1 (48 bytes) and .data_input2 (24 bytes).



```
text   rodata   data   bss   total
 148        0     72    24     244  corr.eln
 148        0     72    24     244  Total
```

a) Default Output

```
corr.eln:
        0  .default
       48  .data_input1
       24  .data_input2
       24  .data_output
      148  .text
      244  Total
```

b) Long Listing (-l option)

Figure 8-4. Size Utility Output

Appendix A ASCII Character Codes

Decimal	Hex	ASCII
0	00	NUL
1	01	SOH
2	02	STX
3	03	ETX
4	04	EOT
5	05	ENQ
6	06	ACK
7	07	BEL
8	08	BS
9	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	S0
15	0F	S1
16	10	DLW
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US
32	20	SP
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	’
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	‘
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

Appendix B
Assembler Messages

Assembler messages are grouped into four categories:

- Command-Line Errors

  These errors indicate invalid command line options, missing filenames, file open errors, or other invocation errors. Command line errors generally cause the assembler to stop processing.

- Warnings

  Warnings notify the programmer of suspect constructs, but do not otherwise affect the object file output.

- Errors

  These errors indicate problems with syntax, addressing modes, or usage. In these cases the resulting object code is generally not valid.

- Fatal Errors

  Fatal errors signify serious problems encountered during the assembly process such as lack of memory, file not found, or other internal errors. The assembler halts immediately.

The assembler also provides information on the source field location of the error, if it can be ascertained. If a listing file is produced, messages ordinarily appear immediately before the line containing the error. One exception is when the relationship between the first and last instructions in a DO loop produces an error. In this case, the error text appears after the last instruction at the end of the loop. Messages are always routed to standard output.

B.1 Assembler Command-Line Errors

Cannot open command file
Cannot open listing file
Cannot open object file

The file associated with a `-f`, `-l`, or `-b` command line option was not found.

Cannot open source file

The assembly source input file was not found.

Duplicate listing file specified - ignored
Duplicate object file specified - ignored

The `-l` and `-b` command line options were encountered more than once on the command line. Only the first occurrence of each option is used. The assembler continues processing.

Illegal command line -D option argument

The symbol name given in a `-d` command line option is invalid (possibly too long or does not begin with an alphabetic character), or the substitution string is not enclosed in single quotes.

Illegal command line -I option argument

A problem occurred when attempting to save the include file path string.

Illegal command line -M option argument

A problem occurred when attempting to save the MACLIB file path string.

Illegal command line option

The option specified on the command line was not recognized by the assembler.

Interrupted

The assembler was interrupted by a keyboard break (Control-C).

LDB option with no listing file specified; using source file

The LDB option was given on the command line without a corresponding `-l` to generate a listing file. If a listing file is not produced, debugging packages cannot use it for source tracking. Therefore the assembler uses the default assembly language file as input for source tracking.

Missing command line option argument

The expected arguments following a command line option specifier were missing.

Missing source filename

There must be at least one source filename specified on the command line.

Source file name same as listing file name
Source file name same as object file name

One of the source files appeared to the assembler to have the same name as the specified listing or object file. The assembler aborts rather than potentially writing over a source input file.

B.2 Assembler Warnings

A2: AGU register contents may be affected by register write in previous cycle

Absolute address too large to use I/O short - long substituted

The absolute address is not within the range specifying an I/O short address, even though the I/O short forcing operator has been used. The assembler substitutes long absolute addressing.

Absolute address too large to use short - long substituted

The absolute address value being forced short will not fit in the storage allocated for a short address. The assembler substitutes long absolute addressing.

Absolute address too small to use I/O short - long substituted

The absolute address is not within the range specifying an I/O short address, even though the I/O short forcing operator has been used. The assembler substitutes long absolute addressing.

Automatic loop alignment occurred — *x* nops added prior to loop

The *x* is a number representing the number of nop padding added.

Cannot force short addressing for source and destination

In a MOVEP or MOVES instruction an attempt was made to force both operands to short or I/O short. The second operand defaults to long.

CL2: At least 7 cycles required between PCTL writes or STOP/WAIT

Debug directives ignored - use command line debug option

A source-level debug directive was encountered but the assembler command line `-g` option was not given.

Destination operand assumed I/O short

Neither operand in a **MOVEP** instruction is explicitly declared I/O short; however, the source operand does not qualify, so the destination operand is assumed to be the I/O short operand.

Directive not allowed in command line absolute mode

The **MODE** directive is ignored when the assembler command line `-a` option is active.

Duplicate listing file specified - ignored
Duplicate object file specified - ignored

The `-b` or `-l` command line options were given more than once.

Explicit bottom margin ignored with page length of zero
Explicit top margin ignored with page length of zero

The top or bottom margin parameters to a PAGE directive are ignored because the page length parameter was zero.

Expression value outside fractional domain

The expected fractional value was not within the range, $-1.0 \leq m < 1$.

Immediate value too large to use short - long substituted

An immediate data value being forced short is too large to fit in the space allocated for a short immediate value. The assembler substitutes long immediate addressing.

Improper nesting of DO loops

The end address of a subordinate (nested) DO loop is greater than or equal to the end address of the loop enclosing it. The end address of a nested DO instruction must be less than the end address of the enclosing loop.

Instruction cannot appear in interrupt vector locations

Some instructions cannot be used reliably as interrupt code.

Instruction does not allow data movement specified - using MOVE encoding
Instruction does not allow data movement specified - using MOVEP encoding
Instruction does not allow data movement specified - using MOVES encoding

An inappropriate MOVE-type instruction was written given the type of the operands. The assembler substitutes a valid encoding for the operands in question.

I/O short absolute address cannot be forced

I/O short absolute addressing is not valid for this operation. An appropriate addressing mode (long absolute, short jump, short absolute) is substituted.

I/O short absolute address cannot be forced - long substituted

I/O short absolute addressing is not valid for this operation. The assembler substitutes long absolute addressing.

I/O short address cannot be forced

I/O short addressing is not valid for this operation. An appropriate addressing mode (long, short, short jump) is substituted.

I/O short address cannot be forced - long substituted

I/O short addressing is not valid for this operation. The assembler substitutes long addressing.

Label field ignored

The assembler directive does not allow a label, so the assembler will not store the label value in the symbol table.

LN3: Consecutive DOEN without intervening loopstart

Load location counter overflow
Load location counter underflow

The load location counter exceeded its maximum or minimum value. The assembler wraps the counter value around and continues.

Load reserved address space violation

The load location counter has incremented into a reserved area of data address space.

Long absolute address cannot be forced

Long absolute addressing is not valid for this operation. An appropriate addressing mode (I/O absolute, short jump, short absolute) is substituted.

Long absolute address cannot be forced - substituting I/O short addressing

Long absolute addressing is not valid for this operation. The assembler substitutes I/O short addressing.

Long absolute address cannot be forced - substituting short addressing

Long absolute addressing is not valid for this operation. The assembler substitutes short absolute addressing.

Long immediate cannot be forced

Long immediate data is not valid for this operation. An appropriate size for the target DSP is substituted.

Long PC-relative address cannot be forced

Long PC-relative addressing is not valid for this operation. An appropriate addressing mode (short PC-relative) is substituted.

Macro expansion not active

A macro must have been called prior to using the @ARG() or @CNT() functions.

Macro name is the same as existing assembler directive
Macro name is the same as existing assembler mnemonic

The name of the macro being defined conflicts with the name of an assembler directive or mnemonic. Either use a different macro name or use the RDIRECT directive to remove the directive or mnemonic name from the assembler lookup tables.

No control registers accessed - using MOVE encoding

A MOVEC-type instruction was given but no control registers were used as operands. The assembler substitutes a valid encoding for the operands in question.

Number of macro expansion arguments is greater than definition
Number of macro expansion arguments is less than definition

A discrepancy exists between the number of arguments specified in a macro definition and the number of arguments provided in the macro call.

Options for both debug and strip specified - strip ignored

Both the `-g` and `-z` options were given on the command line. The `-g` option takes precedence.

P space not accessed - using MOVE encoding

A MOVEP-type instruction was given but a P memory reference was not used as an operand. The assembler substitutes a valid encoding for the operands in question.

Page directive with no arguments ignored with page length of zero

A PAGE directive with no arguments, which ordinarily produces a form feed in the listing output, is ignored because a previous PAGE directive specified a page length of zero.

PC-relative address too large to use short - long substituted

The PC-relative offset is not within the range specifying a short PC-relative offset, even though the short forcing operator has been used. The assembler substitutes long PC-relative addressing.

Post-update operation will not occur on destination register

If the source operand in a MOVE operation specifies a post-update addressing mode and the destination register is the same as the source operand register then the post-update operation will not take place.

PRCTL directive ignored - no explicit listing file

The PRCTL directive takes effect only if the `-l` option is used on the command line to explicitly specify a listing file.

Redefinition of symbol

A symbol used in a DEFINE directive has been redefined without an intervening UNDEF directive. The assembler discards the previous definition and replaces it with the new definition.

Rounding not available with LMS move - using MAC/MPY encoding

A MACR or MPYR instruction was specified in conjunction with LMS move syntax.

Runtime location counter overflow
Runtime location counter underflow

The runtime location counter exceeded its maximum or minimum value. The assembler wraps the counter value around and continues.

Runtime reserved address space violation

The runtime location counter has incremented into a reserved area of data address space.

Short absolute address cannot be forced

Short absolute addressing is not valid for this operation. An appropriate addressing mode (long absolute) is substituted.

Short absolute address cannot be forced - long substituted

Short absolute addressing is not valid for this operation. The assembler substitutes long absolute addressing.

Short immediate cannot be forced

Short immediate data is not valid for this operation. The assembler substitutes an appropriate size.

Short PC-relative address cannot be forced

Short PC-relative addressing is not valid for this operation. The assembler substitutes an appropriate addressing mode (long PC-relative).

Source operand assumed I/O short

Neither operand in a MOVEP instruction is explicitly declared I/O short; however, the destination operand does not qualify, so the source operand is assumed to be the I/O short operand.

SR2: Instruction may be affected by previous SR change

Storage block size not a power of 2

The expression in a DSR directive did not evaluate to a power of 2. Since the DSR directive is generally used to allocate FFT buffers for reverse carry operations, the size of the buffer may be in error.

String truncated in expression evaluation

Only the first four characters of a string constant are used during expression evaluation except for arguments to the DC directive.

Strip not valid in relocatable mode - ignored

The `-z` option is valid only when the `-a` option is also given.

Unresolved external reference

Ordinarily, the assembler does not flag unresolved references in relative mode, assuming they will be resolved at link time. If the UR option is specified, the assembler will generate this warning if any symbols are undefined during the second pass.

B.3 Assembler Errors

A1: At least two cycles are required between MCTL modification and address pointer usage
A1: MCTL register write and R register usage not permitted within a group

A2: AGU register contents are not available for an additional cycle

A3: A valid group must follow a group containing JT, JF, or TRAP

Absolute address contains forward reference - force short or I/O short address

The assembler attempted a default to short addressing which failed. Either the absolute address is too large or it needs to be forced I/O short.

Absolute address must be either short or I/O short

The absolute address is too large for a short address and outside the range of valid I/O short addresses.

Absolute address too large to use I/O short
Absolute address too small to use I/O short

The absolute address being forced short is outside the range of valid I/O short addresses. This usually means that the I/O short address has not been ones extended.

Absolute address too large to use short

The absolute address value is too large to be forced short.

Absolute addressing mode not allowed

Absolute operands are not allowed with some instructions, in particular parallel XY data memory moves.

Address mode syntax error - expected ')'
Address mode syntax error - expected '+'
Address mode syntax error - expected '+' or '-'
Address mode syntax error - expected comma
Address mode syntax error - expected comma or end of field
Address mode syntax error - expected offset register
Address mode syntax error - extra characters
Address mode syntax error - probably missing ')'

A syntax error was detected when scanning the source line operand and/or X and Y data fields. These errors may indicate omission of a source operand, insufficient white space between fields, or improper specification of address register indirect addressing modes.

Argument outside function domain

An argument to one of the transcendental built-in functions was inappropriate.

Arithmetic exception

An internal floating point exception occurred while evaluating an expression. The result of the evaluation is probably not valid.

Assembler directive or mnemonic not found

An argument to the RDIRECT directive was not a recognized assembler directive or mnemonic.

Base argument larger than machine word size

The base parameter of a @FLD() function has a value larger than can fit in the target machine word.

Binary constant expected

A character other than ASCII '0' or '1' either followed the binary constant delimiter (**%**) or appeared in an expression where a binary value was expected by default.

Bit mask cannot span more than eight bits

If the first operand of a BFxxx-type instruction was shifted one bit to the right until the low-order bit was a 1, the resulting value must not exceed $FF hexadecimal.

Cannot conditionally repeat write to memory

A move to memory cannot be preceded by a REPcc-type instruction.

Cannot nest section inside itself

A section of a given name may not have another SECTION directive with the same name declared inside it.

Cannot nest symbol definitions

A source-level debug **.DEF** directive was encountered inside another .DEF-.ENDEF pair.

Cannot open include file

The specified INCLUDE file cannot be found, or the operating system limit on open files has been exceeded.

CL1: Parallel PCTL access not allowed in a group

CL2: At least 7 cycles required between PCTL writes or STOP/WAIT

CL3: PCTL write in a group with STOP/WAIT not allowed

CONST option must be used before any label

This option, which prevents EQU symbols from being exported to the object file, must be given before any label is encountered in the source file.

Contents of assigned register in previous instruction not available

Due to pipelining, if an address register (Rn or Nn) is changed in the previous instruction, the new contents are not available for use as a pointer until the next instruction.

Count must be an integer value

The argument to a DUP directive did not evaluate as an integer expression.

CRE option must be used before any label

The CRE option must be activated before any labels are encountered so that the assembler can append cross-reference data to all applicable symbol table entries.

D1: Illegal instruction in delay slot

D2: Instruction which uses the SR is not allowed in RTED delay slot

D3: RTE/D and instruction which uses SR not permitted in same group

D4: Instruction is not allowed in RTSD/RTED/RTSTKD delay slot

D5: Instruction is not allowed in JSRD delay slot
D5: JSR/D and moves to SR are not permitted in the same group

Data allocation exceeds buffer size

Data allocated between a BUFFER-ENDBUF sequence exceeded the size specified in the BUFFER directive.

Decimal constant expected

A character other than ASCII 0 through 9, either followed the decimal constant delimiter (‘) or appeared in an expression where a decimal value was expected by default.

DEFINE symbol must be a global symbol name

A local label (a symbol beginning with the underscore character) may not be used as a DEFINE directive symbol.

Displacement address mode not allowed

Long displacement addressing is not allowed with some instructions, in particular parallel XY data memory moves.

Divide by zero

The expression evaluator detected a divide by zero.

DO loop address must be in current section

The loop address of a DO loop cannot fall outside the bounds of its enclosing section. This is particularly important in relative mode as the loop address is calculated based on the starting address of the section.

Dummy argument not found

The dummy argument name given as an argument to the @ARG() function was not found in the macro dummy argument list.

Duplicate destination register not allowed

If the opcode-operand portion of an instruction specifies a destination register, the same register or portion of that register may not be specified as a destination in the parallel data bus move operation.

Duplicate destinations in paired instructions

Duplicate source and destination register not allowed

If the opcode-operand portion of an instruction specifies a source register, the same register or portion of that register may not be specified as a destination in the parallel data bus move operation.

ELSE without associated IF directive

An ELSE directive was encountered without a preceding IF conditional assembly directive.

Empty bit mask field

The first operand of a BFxxx-type instruction was zero.

ENDBUF without associated BUFFER directive

An ENDBUF directive was encountered without a preceding BUFFER directive.

.ENDEF without associated .DEF directive

A source-level debug .ENDEF directive was encountered without a preceding .DEF directive.

ENDIF without associated IF directive

An ENDIF directive was encountered without a preceding IF conditional assembly directive.

ENDM without associated MACRO directive

An ENDM directive was encountered without a preceding MACRO directive.

ENDSEC without associated SECTION directive

An ENDSEC directive was encountered without a preceding SECTION directive.

EQU requires label

The EQU directive must have a label to associate with the equated expression.

Error in mnemonic table

EXITM without associated MACRO directive

An EXITM directive was encountered without a preceding MACRO directive.

Expression cannot have a negative value

Some directives do not allow negative expression arguments, as for example in the PAGE directive controls.

Expression contains forward references

Some directives do not allow expression arguments which have not yet been defined in the source, as for example in the IF, EQU, or SET directives.

Expression must be greater than zero

Some directives require a nonzero argument, as for example in the BSC directive.

Expression result must be absolute

Certain directives and some assembler usage require absolute values as arguments or operands.

Expression result must be integer

Certain directives and some assembler usage require integer values as arguments or operands.

Expression result too large

The expression evaluated to a value greater than the acceptable range. This error can occur when an expression result exceeds the native word size of the target DSP.

External reference not allowed in expression

References to external symbols (e.g., symbols not defined in the current assembly source input) are not allowed in some types of byte or integer expressions.

External reference not allowed in function

References to external symbols (e.g., symbols not defined in the current assembly source input) are not allowed as direct or indirect arguments to any built-in function.

Extra characters beyond expression

The expression evaluator found extra characters after the end of a valid expression. Unbalanced parentheses can cause this error.

Extra characters following string

An end-of-string delimiter was followed by unexpected characters on the source line.

Extra characters following symbol name

A non-alphanumeric character other than the underscore (_) was encountered in a symbol name.

Extra characters in function argument or missing ')' for function

Mismatched parentheses or wrong number of parameters in a function invocation.

Extra characters in operand field

The PAGE directive contains too many operands.

Extra fields ignored

There were extra fields specified in an assembler directive.

First data move destination accumulator same as operand destination accumulator

The destination of the data move field is the same as the Data ALU destination.

First data move source accumulator same as operand destination accumulator

The source of the data move field is the same as the Data ALU destination.

Floating point constant expected

A character other than ASCII 0 through 9, e or E, or “.'” appeared in an expression where a floating-point value was expected by default.

Floating point not allowed in relative expression

Relative expressions are generally used for address computation, therefore a floating-point value would not be appropriate.

Floating point value not allowed

An immediate value expressed in floating-point notation is only valid in a MOVE-type instruction.

Forcing not specified

The type of forcing operand was not given in a FORCE directive.

Function result out of range

The result computed by a transcendental function was too large to be represented on the host machine.

GG1: Too many total instructions

GG2: Total instruction length cannot exceed 8 words (including prefix)

GG3: Too many total DALU ops plus BFU ops
GG3: Only one bit mask instruction permitted in group

GG4: Only one t bit update instruction permitted in group
GG4: Only one SP update instruction permitted in group
GG4: Only one address register update permitted in group
GG4: Double push may only include one register from eeeee or EEEEE
GG4: Double pop may only include one register from eeeee or EEEEE
GG4: Only one change of flow permitted in group

GG5: DALU register may only be used four times per set

GL option must be used before any section

The GL option must be activated before any explicit sections are encountered so that the assembler can insure that all section symbols are global.

GLOBAL without preceding SECTION directive

A GLOBAL directive was encountered outside any previously defined section.

GP1: Too many extension words in paired grouping

GP3: Only one change of flow permitted in group
GP3: Only one di or ei permitted in group
GP3: Only one stop or wait permitted in group
GP3: Only one debug instruction permitted in group
GP3: Only one mark instruction permitted in group

GP3A: DOEN type instruction may not be grouped with break

GP4: RTE/D and AGU or IFT/F/A combination not permitted in same group

GP5: N or M used as source reg more than once in a group

GS option must be used before any section

The GS option must be activated before any explicit sections are encountered so that the assembler can use the appropriate counters for section relocation.

GS: Pairing of non-pairable instructions

Hex constant expected

A character other than ASCII 0 through 9, a through f, or A through F either followed the hexadecimal constant delimiter ($) or appeared in an expression where a hexadecimal value was expected by default.

IC option must be used before any symbol, section, or macro definition

The IC option must be activated before any symbols, sections, or macros are defined so that the assembler can remain consistent when storing label names in the symbol table.

IDENT directive must contain revision number
IDENT directive must contain version number

The version and revision numbers are both required arguments for the IDENT directive.

Illegal directive in buffer declaration

A directive was encountered between a BUFFER-ENDBUF pair that is not allowed in that context. Some invalid directives include any other buffer-type directive (DSM, DSR, etc.), section directives, or any directive which alters the current location counter designation (MODE, ORG).

Illegal directive inside .DEF-.ENDEF declaration
Illegal directive outside .DEF-.ENDEF declaration

Some source-level debug directives, such as .FILE, make no sense and are not allowed inside .DEF-.ENDEF declarations. Conversely, other directives such as .VAL are not allowed outside of a .DEF-.ENDEF declaration.

Illegal directive inside DO loop

A directive was encountered inside a DO loop that is not allowed in that context. Some invalid directives include any buffer-type directive (DSM, DSR, etc.), section directives, or any directive which alters the current location counter designation (MODE, ORG).

Illegal function argument

An invalid argument was passed to one of the assembler built-in functions, in particular the @LCV() function.

Illegal instruction in single-instruction DO loop

A conditional break instruction (BRKcc) cannot be used as the only instruction in a DO loop.

Illegal memory counter specified

The memory counter designation supplied in the ORG directive was not one of H (high), L (low), or a positive integer expression in parentheses.

Illegal memory map character

The memory map character supplied in the ORG directive was not one of I (internal), E (external), R (ROM), A (port A), or B (port B).

Illegal memory space specified

The memory space given is either invalid or inappropriate for the desired operation.

Illegal move field destination specified
Illegal move field destination register specified

The destination operand in a data memory move is invalid for the type of instruction specified.

Illegal move field source specified

The source operand in a data memory move is invalid for the type of instruction specified.

Illegal operator for floating point element

Bitwise operators are invalid for floating point values.

Illegal option

An argument to the OPT directive is invalid.

Immediate addressing mode not allowed

Immediate operands are not allowed with some instructions, in particular program memory moves (**MOVEM**).

Immediate operand not allowed

Immediate operands are not allowed with some instructions, in particular program memory moves (MOVEM).

Immediate value too large

The immediate operand value is too large for the space allotted in the instruction.

Immediate value too large to use short

The immediate value being forced short is too large to fit into the instruction word.

Increment value cannot be zero

The increment parameter to a DUPF directive must be greater than zero.

Initial debug directive must be .FILE

In a source file containing debug directives being assembled with the -G option the .FILE directive must be the first source-level debug directive in the input stream.

Instruction cannot appear immediately after control register access

Some instructions must not appear immediately after certain control registers have been accessed.

Instruction cannot have ift, iff, ifa in one packet

Instruction could not be reordered for mux

Instruction does not allow data movement specified

The desired operation may only be done with a MOVE instruction.

Invalid address expression

An attempt was made to evaluate an expression consisting of two relative terms with the same sign.

Invalid addressing mode

The addressing mode of one of the operands in the instruction was not recognized.

Invalid buffer type

The buffer type specified in a BADDR or BUFFER directive was not one of M (modulo) or R (reverse-carry).

Invalid conditional register transfer syntax

The syntax for an IFcc or FFcc conditional address register move was incorrect.

Invalid destination register

The first data move destination register in a double memory read operation was not valid.

Invalid dummy argument name

Macro argument names cannot be local symbols, e.g. they cannot begin with the percent (**%**) character.

Invalid function name

The name following the function invocation character (**@**) was not recognized.

Invalid label field width specified

The argument given to the LSTCOL directive does not allow enough room on the listing line for the remaining fields to be output.

Invalid macro name

Macro names cannot be local symbols, e.g. they cannot begin with the percent (**%**) character.

Invalid memory space attribute

The memory space attribute given is not the letter P.

Invalid mode

The mode specified in a MODE directive was not RELATIVE or ABSOLUTE.

Invalid opcode field width specified
Invalid operand field width specified

The argument given to the LSTCOL directive does not allow enough room on the listing line for the remaining fields to be output.

Invalid page length specified

The minimum page length allowed by the PAGE directive is 10 lines per page. The maximum is 255.

Invalid page width specified

The minimum page width allowed by the PAGE directive is 1 column per line. The maximum is 255.

Invalid radix expression

The expression in the RADIX directive does not evaluate to one of the supported constant bases (2, 8, 10, or 16).

Invalid register combination

The source operand registers in a FMPY instruction cannot be used together.

Invalid register specified

The direct register operand is incorrect for this instruction.

Invalid relative expression

The terms of a relative expression may only participate in addition and subtraction operations and must have opposing signs.

Invalid section directive modifier

The qualifier specified in a SECTION directive was not GLOBAL.

Invalid section name

Section names cannot be local symbols, e.g. they cannot begin with the percent (**%**) character.

Invalid shift amount

A shift expression must evaluate to within the range 0 <= n <= *m*, where *m* is the maximum address of the target DSP.

Invalid source address mode

The source address mode in a MOVEP **i**nstruction was not valid.

Invalid source address register
Invalid source register

The source register in a double memory read operation was not valid.

Invalid storage class

The storage class given in a source-level debug symbol declaration is unknown.

Invalid tabs stops specified

The argument to the TAB directive is out of range.

I/O short addressing mode not allowed

An operand was forced I/O short when I/O short addressing was not allowed.

LB option must be used before any code or data generation

The LB option must be specified before any code or data in order for the assembler to increment the location counter appropriately.

LC1: Branch to last two execution sets in a loop not allowed

LC2: Change of flow not allowed between LA and LA-2

LC3: No conditional branch is allowed in the execution set before the start address of a short loop

LC4: Illegal sequence: T bit modification followed by conditional jump, followed by loop start address

LC5: No conditional branches allowed in the last four execution sets of a long loop

LC6: No conditional jump at loop address-3 if T bit modification precedes the jump instruction

LC7: Destination address of SKIP/BREAK/CONT cannot be in the same loop

LC8: No change of flow instruction allowed in short loops

LC9: SKIPS/BREAK/CONT target cannot be followed by consecutive loop addresses

LC10: JSR/BSR to LA-2 of a long loop or SA of a short loop

LD1: SKIPLS instruction not allowed immediately after DOEN/SH or move to LC

LD2: Three, four, or more execution sets are required between DOEN or write to LC and loop end

LD3: Two, three, or more execution sets are required between DOENSH or write to LC and the first execution set

LD4: Write to SR is not allowed before DOEN/SH

LD5: One, two or more execution sets required between LC update and CONT/D instruction

LD6: At least three execution sets required between DOSETUP and last execution set

LD7: At least one execution set required between CONT/D and modification of SA

LD8: At least three execution sets required between read of LC and last set of loop

LD9: At least one execution set required between read of LC and first set of short loop

LDB option must be used before any code or data generation

The LDB option must be specified before any code or data in order for the assembler to establish the debug source file appropriately.

Left margin exceeds page width

The blank left margin value in the PAGE directive exceeds the default or specified page width parameter.

Length value greater than string size

The length parameter in a substring construct is larger than the composite length of the input string argument.

Line too long

Source statements, including continuation lines, cannot exceed 512 characters in length.

LG3: Less than three sets between MOVE/PUSH SR and end of loop

LG4: Short loop SA follows MOVE/PUSH SR

LL1: Illegal delay slot instruction in last two execution sets of a loop

LL2: DOEN/SH or write to LC not allowed in last three execution sets of a loop

LL3: Illegal delay slot instruction in short loop

LN1: Nested loops cannot end in same address

LN2: A loop may only be nested inside a loop of a smaller DOEN number

LN4: Nested short loopend cannot occur in last two execution sets of outer loop

LOC option must be used before any local label

The LOC option must appear before any local label so that the assembler can keep the local label lists synchronized.

Local symbol names cannot be used with GLOBAL

Percent (%) labels are not allowed with this directive.

Long absolute address cannot be used

An operand was forced long where only a short or I/O short address was valid.

Long absolute cannot be used - force short or I/O short

A forward reference was forced long where only a short or I/O short address was valid.

loopstart, loopend mismatch

Macro cannot be redefined

A macro name cannot be used as the label for a second macro definition in the same source file unless the macro is defined and used within a declared section.

Macro not defined

The macro name was not found in the macro lookup table.

Macro value substitution failed

The evaluation of a macro argument expression failed.

Memory bounds greater than maximum address

The bounds argument in a LOMEM or HIMEM directive is invalid.

Memory counter designator value too large

The integer counter designator in an ORG directive is greater than 65535.

Memory space must be P or NONE

An END directive was encountered while the runtime memory space was not P or N.

Missing '(' for function

All assembler built-in functions require at least one argument which must be enclosed in parentheses.

Missing ')' in expression

Parentheses are not balanced in an expression.

Missing argument

The argument to a DUPA or DUPC directive was not found.

Missing definition string

The substitution string for a DEFINE directive is missing.

Missing delimiter in substring

A substring construct was missing the closing square bracket.

Missing expression

An expression was expected by the expression evaluator.

Missing filename

No filename was provided as an argument to the INCLUDE directive.

Missing macro name

A MACRO directive was encountered without a label, or the macro name was omitted from a PMACRO directive.

Missing option

The OPT directive was specified without an argument.

Missing or mismatched quote

A single or double quote character was expected by the string parsing routines.

Missing pathname

No pathname was provided as an argument to the MACLIB directive.

Missing quote
Missing quote in string

A single or double quote character was expected by the string parsing routines.

Missing section name

No section name was given as an argument to the SECTION directive.

Missing string after concatenation operator

The string concatenation operator (++) must be followed by another quoted string.

Mode not specified

The MODE directive was not followed by either RELATIVE or ABSOLUTE.

MU option must be used before any code or data generation

The MU option must be given before any data allocation directive (BSC, DC, DS, DSR) or instruction appears in the source file.

Negative immediate value not allowed

The immediate count value for a DO or REP instruction cannot be less than zero.

Negative or empty DO loop not allowed

The loop address given in a DO instruction must specify an address at least one greater than the current program counter value.

No instructions in enclosing blocks

Not enough fields specified for instruction

There were no operands specified for a MOVE, MOVEC, MOVEM, or MOVEP instruction.

Offset value greater than string size

The offset parameter in a substring construct is larger than the composite length of the input string argument.

Only absolute addressing allowed

The instruction allows only absolute addressing.

Only absolute and register direct addressing allowed

The instruction allows only absolute and register direct addressing.

Only immediate addressing allowed

The instruction allows an immediate source operand only.

Only immediate and register direct addressing allowed

The instruction allows only immediate and register direct addressing modes.

Only immediate and register direct and indirect addressing allowed

The instruction allows only immediate, register direct, and register indirect addressing modes.

Only one P memory move instruction permitted in group

Only PC-relative addressing allowed

The instruction allows only PC-relative addressing.

Only PC-relative and register direct addressing allowed

The instruction allows only PC-relative and register direct addressing.

Only post-increment or post-increment by offset addressing allowed

Moves to P memory allow only post-increment or post-increment by offset addressing.

Only register direct addressing allowed

The instruction allows only register direct addressing.

Only register direct and indirect addressing allowed

The instruction allows only register direct and indirect addressing.

Only register indirect addressing allowed

The instruction allows only register indirect addressing.

Operation not allowed with address term

Only addition and subtraction are allowed in expressions involving addresses or relative terms.

Page length too small for specified top and bottom margins

The sum of the top and bottom margins specified in the PAGE directive is greater than the page length - 10.

Page length too small to allow default bottom margin

The bottom margin exceeds the page length specified in the PAGE directive.

PC-relative address too large to use short

The PC-relative offset being forced short is too large to fit into the instruction word.

PC-relative addressing mode not allowed

The PC-relative addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Phasing error

The value associated with a symbol has changed between Pass 1 of the assembly and Pass 2. This error can occur spontaneously in conjunction with other errors. The assembler is designed to avoid phasing errors in general. If a phasing error occurs without any other errors, this may represent an internal error which should be reported to StarCore.

One exception is the use of the checksumming function @CHK() with the EQU directive. Instruction encoding may be incomplete after the first pass due to forward referencing, causing the checksum value to change between passes. Because of this, the SET directive must be used to assign the checksum value to a symbol.

Possible invalid white space between operands or arguments

The assembler verifies that fields which should not contain operands or values are empty. If these fields are not empty, the assembler produces this error.

Post-decrement addressing mode not allowed

The post-decrement addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-decrement by offset addressing mode not allowed

The post-decrement by offset addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-increment addressing mode not allowed

The post-increment addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Post-increment by offset addressing mode not allowed

The post-increment by offset addressing mode is not allowed for this instruction. The restriction applies, for example, to bit manipulation instructions and some jump-type instructions.

Pre-decrement addressing mode not allowed

The pre-decrement addressing mode is not allowed for this instruction. The restriction applies, for example, to instructions which include parallel XY memory data transfers.

R2: Cannot compare a register to itself

RDIRECT directive not allowed in section

Since the effect of the RDIRECT directive is global, it cannot be used within a section which has been declared using the SECTION directive. Move the RDIRECT directive outside the declared section to avoid this error.

Redefinition would overflow line

A substitution string declared using the DEFINE directive will cause the current source line to overflow if substitution occurs.

Reference outside of current buffer block
Reference outside of current overlay block

Reference was made to an underscore local label which fell outside the current buffer or overlay definition.

Register direct addressing not allowed

Register direct addressing mode is not allowed for this instruction.

Register displacement valid only with address register R2

Only address register R2 is valid as a displacement register.

Relative equate must be in same section

An EQU directive with a relative expression operand must be defined in the same section as the section associated with the operand expression.

Relative expression must be integer

A relative expression must evaluate to an integer value.

Relative expression not allowed

Relative expressions are not allowed as arguments to the assembler built-in functions.

Relative SET must be in same section

A SET directive with a relative expression operand must be defined in the same section as the section associated with the operand expression.

Relative terms from different sections not allowed

Relative terms defined in different sections are not allowed in expressions. This is because the relationship between the terms is based on enclosing section’s location in memory.

Reserved name used for symbol name

One of the DSP register names has been used as a label, operand, or directive argument. These register names, in either upper or lower case, are reserved by the assembler.

Runtime space must be P

An instruction was encountered and the runtime memory space was not set to P (Program).

Section not encountered on pass 1

The section declared in a SECTION directive was not encountered during the first pass of the assembler. This situation indicates an internal assembler error and should be reported to StarCore.

SET requires label

The SET directive must have a label in order to associate the directive argument with a symbol name.

SET symbol names cannot be used with XDEF

A symbol defined using the SET directive cannot be exported from a section using GLOBAL.

Short absolute address too large

The flagged operand value is greater than the maximum short address of the target DSP.

Short I/O absolute address too large
Short I/O absolute address too small

The flagged operand value is outside the I/O address range of the target DSP.

Short or I/O short address expected

A short or I/O short address was expected as the second operand.

Short PC-relative address too large

The flagged operand value is greater than the maximum PC-relative address of the target DSP.

SR2: Instruction not allowed within two execution sets of an SR change

SR3: Instruction not allowed after SR change

Start argument greater than machine word size

The start parameter of a @FLD() function has a value larger than can fit in the target machine word.

Start position greater than source string size

The start parameter in a @POS() function is larger than the total length of the source string argument.

Storage block size must be greater than zero

The size of a buffer allocated with the DSM, DSR, BSM, BSB, and other buffer directives was too small.

Storage block size out of range

The size of the buffer in a DSM, DSR, BSM, BSB, or other buffer directive is too large to be allocated.

Storage block too large

The runtime location counter overflowed while the assembler was attempting to allocate storage through a DSR directive. The assembler automatically advances the program counter to the next valid base address given the size of the modulo or reverse carry buffer. This error occurs when the sum of the expression in the DSR directive and the runtime location counter value exceed available memory in the current memory space.

SVO option must be used before any code or data generation

The SVO option must be given before any data allocation directive (BSC, DC, DS, DSM, DSR) or any instruction appears in the source file.

Symbol already defined as GLOBAL

The symbol used in an GLOBAL directive has already been defined in a previous directive of the same type.

Symbol already used as SET symbol

The label has already been used in a SET directive. A symbol defined with SET cannot be redefined except through another SET directive.

Symbol cannot be set to new value

The label has been defined previously other than with the SET directive. Only symbols defined using the SET directive may be redefined.

Symbol defined in current section before GLOBAL directive

The GLOBAL directive must appear within a section prior to the definition of any symbols in its argument list. Any symbols within a section which must be accessible outside the section should be declared in a GLOBAL directive immediately following the SECTION directive.

Symbol name too long

Symbols are limited to 512 characters. The first character must be alphabetic, the percent character, or the underscore character (A-Z, a-z, %,_). The remaining characters must be alphanumeric, including the underscore character (A-Z, a-z, 0-9, _).

Symbol not previously defined

The symbol specified in an UNDEF directive was not previously defined in a DEFINE directive.

Symbol redefined

The symbol has already been used as a label in a previous context.

Symbol tag mismatch

A matching tag reference could not be found for a tagged symbol table entry.

Symbol undefined on pass 2

The symbol used as an operand or directive argument was never defined in the source program.

Symbols must start with alphabetic character

Symbol names must begin with an upper or lower case alphabetic character, the percent character (%), or the underscore character (**_**).

SYMOBJ symbol must be a global symbol name

Arguments to the **SYMOBJ** directive cannot be preceded by an underscore.

Syntax error - expected '):'

In an ORG directive using numeric counter designations the parenthesis/colon pair separating the load or runtime address from the memory space, counter, or mapping characters was not found.

Syntax error - expected ':'

In an ORG directive, the colon separating the load or runtime address from the memory space, counter, or mapping characters was not found.

Syntax error - expected '>'

The closing angle bracket in a non-local INCLUDE directive argument was not found.

Syntax error - expected comma

The comma separating operands in an instruction or directive was not found.

Syntax error - expected quote

The assembler expected the start of a quoted string.

Syntax error - extra characters

Extra characters were found after an instruction or directive operand.

Syntax error - missing address mode specifier

An instruction operand was not specified.

Syntax error in dummy argument list

A character other than a comma was found separating the dummy arguments in a macro definition (MACRO directive), or a dummy argument began with the percent character (**%**).

Syntax error in macro argument list

A character other than a comma was found separating the arguments in a macro call.

Syntax error in macro name list

A character other than a comma was found separating the arguments in a PMACRO directive name list.

Syntax error in symbol name list

A character other than a comma was found separating the arguments in an XDEF or XREF directive name list.

T1: IFc is not allowed to follow a group containing a T bit modification

Tag name not found

A matching tag name could not be found for the current source-level debug structure or union declaration.

Too many fields specified for instruction

An instruction field that was expected to be empty contained data other than a comment.

Too many paired MOVE ops

Two dummy arguments are the same

Two dummy arguments in a macro definition (MACRO directive) have the same name.

UNDEF symbol must be a global symbol name

The argument to an UNDEF directive cannot be a local label.

Unexpected end of file - missing COMMENT delimiter

The second occurrence of the delimiter character in a COMMENT directive was never found.

Unexpected end of file - missing ENDBUF

A BUFFER directive was encountered without a closing ENDBUF **d**irective.

Unexpected end of file - missing ENDIF

An IF directive was encountered without a closing ENDIF directive.

Unexpected end of file - missing ENDM

A macro definition was started using the MACRO directive, but the end of the source file was encountered before a closing ENDM directive was found.

Unexpected end of file - missing ENDSEC

A SECTION directive was found without a closing ENDSEC directive.

Unknown math error

A transcendental math function returned an error that could not be classified as out of range or outside the function domain.

Unrecognized mnemonic

A symbol in the assembler opcode field was not a defined macro, an instruction mnemonic, or a directive.

Unrecognized secondary mnemonic

A symbol in the assembler secondary opcode field was not one of the instructions FADD, FSUB, or FADDSUB.

V1: Offset + Width > 40 bits

V2: Offset + Width > 40 bits

V3: SP must be a multiple of eight

Value argument larger than machine word size

The value parameter of a @FLD() function has a value larger than can fit in the target machine word.

Width argument greater than machine word size

The width parameter of a @FLD() function has a value larger than can fit in the target machine word.

XDEF without preceding SECTION directive
XREF without preceding SECTION directive

An XDEF or XREF directive was encountered outside any previously defined section.

XLL option must be used before any local label

The XLL option must be activated before any local labels are encountered so that the assembler can make the appropriate entries in the symbol table.

XR option must be used before any label

The XR option must be activated before any labels are encountered so that the assembler can make the appropriate entries in the symbol table.

B.4 Assembler Fatal Errors

<mode> encoding failure

A bad address mode indicator or register number was passed to the assembler encoding routines. <mode> represents the register set or addressing mode in question. This is a serious internal error that should be reported to StarCore.

Absolute mode select failure

The mode indicator passed to the absolute addressing mode selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Arithmetic exception

An internal floating-point exception occurred while evaluating an expression. The assembler cannot continue.

Cannot encode instruction
Cannot encode branch instruction
Cannot encode jump instruction

The correspondence between the source opcode mnemonic and the internal opcode type has been corrupted. This is an internal error that should be reported to StarCore.

Cannot seek to start of line number entries
Cannot seek to start of object data
Cannot seek to start of object file
Cannot seek to start of relocation entries
Cannot seek to start of section headers
Cannot seek to start of string table
Cannot seek to start of symbol table

An I/O error occurred which prevented the assembler from positioning correctly in the output object file.

Cannot write file header to object file
Cannot write line number entries to object file
Cannot write optional header to object file
Cannot write relocation entries to object file
Cannot write section headers to object file
Cannot write string table to object file
Cannot write symbols to object file

An I/O error occurred which prevented the assembler from writing data to the output object file.

Cannot write control string to listing file
Cannot write left margin to listing file
Cannot write new line to listing file

Cannot write new page to listing file
Cannot write page header to listing file
Cannot write string to listing file

An I/O error occurred which prevented the assembler from writing data to the output listing file.

Compare select error

The comparison indicator passed to the evaluator selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Debug symbol type failure

The symbol type indicator passed to the debug selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Directive select error

The directive indicator passed to the directive selection logic was not valid. This is a serious internal error that should be reported to StarCore.

DO stack out of sequence

The assembler maintains an internal stack representing DO loop nesting levels. The internal stack pointers have been corrupted.

Error in mnemonic table

The indicator passed to the instruction processing logic was not valid. This is a serious internal error that should be reported to StarCore.

Expression operator failure

Expression operator lookup has failed. This is a serious internal error that should be reported to StarCore.

Expression stack underflow

An attempt has been made to free an expression when there are none to be freed. This is an internal error that should be reported to StarCore.

Fatal segmentation or protection fault
Contact StarCore DSP Operation

A program error has caused the assembler to access an invalid host system address. This generally indicates a bug in the assembler software.

File info out of sequence

File debug information is scrambled. This is a serious internal error that should be reported to StarCore.

File not encountered on pass 1

The file in the source input list was never processed by the assembler during Pass 1. This is an internal error that should be reported to StarCore.

Immediate mode select error

The mode indicator passed to the immediate addressing mode selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Input mode stack out of sequence

The stack for recording whether input is from a file or a macro expansion has been corrupted. This is an internal error that should be reported to StarCore.

Invalid DO loop range check

The value passed to the end-of-DO-loop verification logic is bad. This is an internal error that should be reported to StarCore.

Invalid instruction class

The saved **MAC-**type instruction class has been corrupted. This is an internal error that should be reported to StarCore.

Invalid tag storage class

The saved tag storage class has been corrupted. This is an internal error that should be reported to StarCore.

I/O error writing data word to object file

An I/O error occurred which prevented the assembler from writing data to the output object file.

Location bounds selection failure

The logic for selecting the appropriate bounds array based on the current memory space has returned a bad value. This is an internal error that should be reported to StarCore.

Option select error

The option indicator passed to the option selection logic (OPT directive) was not valid. This is a serious internal error that should be reported to StarCore.

Out of memory - assembly aborted

There is not enough internal memory to perform dynamic storage allocation. Since the assembler keeps all working information in memory, including the symbol table and macro definitions, there is the possibility that memory will be exhausted if many symbols or macros are defined in a single assembly run.

PC-relative mode select failure

The mode indicator passed to the PC-relative addressing mode selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Register selection failure

The register number passed to the multiply mask selection logic was not valid. This is a serious internal error that should be reported to StarCore.

Section counter sequence failure

The ordering of location counter structures has been corrupted. This is an internal error that should be reported to StarCore.

Section stack mode error

The assembler expected to restore a nested section but found the section list empty. This is an internal error that should be reported to StarCore.

Too many lines in source file

An individual source file contained more than 2**31 lines of code.

Too many sections in module

There is a limit of 255 discrete sections in a given source file.

Index

STAR CORE

BRIGHTER DSP TECHNOLOGY!

How to reach us:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1 (800) 441-2447

Asia/Pacific
Motorola Semiconductors H.K. Ltd., Hong Kong
852-26629298

Japan
Motorola Japan, Ltd., Shinagawa-ku, Japan
81-3-5487-8488

Motorola Fax Back System (Mfax™)
1 (800) 774-1848; RMFAX0@email.sps.mot.com

DSP Helpline
dsphelp@dsp.sps.mot.com

Technical Resource Center
1 (800) 521-6274

Internet
http://www.motorola-dsp.com

Lucent Technologies Microelectronics Group
Internet
http://www.lucent.com/micro/dsp

Email
docmaster@micro.lucent.com

U.S./Canada
Lucent Technologies Microelectronics Group
1-800-372-2447, FAX 610-712-4106
In CANADA: 1-800-553-2448, FAX 610-712-4106

Asia/Pacific Microelectronics Group
Lucent Technologies Singapore Pte. Ltd., Singapore
Tel. (65) 778 8833, FAX (65) 777 7495

China Microelectronics Group
Lucent Technologies (China) Co., Ltd., Shanghai
Tel. (86) 21 6440 0468, ext. 316, Fax (86)21 6440 0652

Japan Microelectronics Group
Lucent Technologies Japan, Ltd., Shinagawa-ku, Japan
Tel. (81) 3 5421 1600, FAX (81) 3 5421 1700

Europe Microelectronics Group Dataline
Tel. (44) 1189 324 299, FAX (44) 1189 328 148

Digital DNA™
from Motorola

microelectronics group

Lucent Technologies
Bell Labs Innovations